REFERENCE GUIDE

SNAP Connect Python Package Manual

Reference Manual for Version 3.4

©2008-2015 Synapse, All Rights Reserved. All Synapse products are patent pending. Synapse, the Synapse logo, SNAP, and Portal are all registered trademarks of Synapse Wireless, Inc.

Doc# 114-003901-004-B000

6723 Odyssey Drive // Huntsville, AL 35806 // (877) 982-7888 // Synapse-Wireless.com

Disclaimers

Information contained in this Manual is provided in connection with Synapse products and services and is intended solely to assist its customers. Synapse reserves the right to make changes at any time and without notice. Synapse assumes no liability whatsoever for the contents of this Manual or the redistribution as permitted by the foregoing Limited License. The terms and conditions governing the sale or use of Synapse products is expressly contained in the Synapse's Terms and Condition for the sale of those respective products.

Synapse retains the right to make changes to any product specification at any time without notice or liability to prior users, contributors, or recipients of redistributed versions of this Manual. Errata should be checked on any product referenced.

Synapse and the Synapse logo are registered trademarks of Synapse. All other trademarks are the property of their owners. For further information on any Synapse product or service, contact us at:

Synapse Wireless, Inc.
6723 Odyssey Drive
Huntsville, Alabama 35806
256-852-7888
877-982-7888
256-924-7398 (fax)

www.synapse-wireless.com

License governing any code samples presented in this Manual

Redistribution of code and use in source and binary forms, with or without modification, are permitted provided that it retains the copyright notice, operates only on SNAP® networks, and the paragraphs below in the documentation and/or other materials are provided with the distribution:

Copyright 2008-2015, Synapse Wireless Inc., All rights Reserved.

Neither the name of Synapse nor the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SYNAPSE AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SYNAPSE OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF SYNAPSE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

1. Overview 1

Introducing SNAP Connect 2

SNAP Connect Licensing 3

2. Installation 4

Sample Application: SimpleMcastCounter 5

Remote Connection Capabilities 6

Sample Application: Remote TCP Access 7

Recommended Practices 8

Hooks 8

Asynchronous Programming 8

Multiple Instances 10

3. Examples: SNAP Connect on the E10 Gateway 11

UserMain.py 11

Sample Application – simpleSnapRouter.py 17

Usage 17

Source Code 19

Source Code with Commentary 21

4. Additional Examples 25

EchoTest – Simple Benchmark 25

SPY File Upload 26

BulkUpload – Uploading SPY files to multiple nodes 27

FirmwareUpgrader – Creating a GUI with wxPython 28

RobotArm – Example SNAP Connect Web Server 29

Hints and Tips 31

5. *SNAP Connect* API Reference – The Functions 34

__init__ 34

accept_sniffer 35

accept_tcp 35

add_rpc_func 36

allow_serial_sharing 36

cancel_upgrade 37

close_all_serial 37

close_serial 37

connect_tcp 37
data_mode 38
directed_mcast_rpc 39
disconnect_tcp 39
dmcast_rpc | dmcastRpc 40
get_info | getInfo 41
load_nv_param | loadNvParam 43
local_addr | localAddr 43
loop 44
mcast_data_mode 44
mcast_rpc | mcastRpc 44
open_serial 45
poll 46
poll_internals 46
replace_rpc_func 47
rpc 47
rpc_alternate_key_used 48
rpc_source_addr | rpcSourceAddr 48
rpc_source_interface 48
rpc_use_alternate_key 48
save_nv_param | saveNvParam 49
set_hook | setHook 49
start_sniffer 50
stop_accepting_sniffer 50
stop_accepting_tcp 51
stop_sniffer 51
traceroute 51
upgrade_firmware 52
vmStat 52

6. *SNAP Connect* API Reference – Constants and Enumerations 54

Constants used with encryption 54
Constants used with close_serial(), open_serial(), HOOK_SERIAL_OPEN and HOOK_SERIAL_CLOSE 54
Constants used with set_hook() 54

Constants used with rpc_source_interface() 55
Constants used with SPY Uploading 55
Constants used with firmware upgrades 55
String Constants used with Logging 56
7. *SNAP Connect* API Reference – NV Parameters 57
ID 0-1 – Reserved for Synapse Use, not used by *SNAP Connect* 57
ID 2 – MAC Address, not used by *SNAP Connect* 57
ID 3 – Network ID, not used by *SNAP Connect* 57
ID 4 – Channel, not used by *SNAP Connect* 57
ID 5 – Multi-cast Processed Groups 57
ID 6 – Multi-cast Forwarded Groups 58
ID 7 – Manufacturing Date, not used by *SNAP Connect* 58
ID 8 – Device Name 58
ID 9 – Last System Error, not used by *SNAP Connect* 58
ID 10 – Device Type, not used by *SNAP Connect* 58
ID 11 – Feature Bits 58
ID 12 – Default UART, not used by *SNAP Connect* 59
ID 13 – Buffering Timeout, not used by *SNAP Connect* 59
ID 14 – Buffering Threshold, not used by *SNAP Connect* 59
ID 15 – Inter-character Timeout, not used by *SNAP Connect* 59
ID 16 – Carrier Sense, not used by *SNAP Connect* 60
ID 17 – Collision Detect, not used by *SNAP Connect* 60
ID 18 – Collision Avoidance, not used by *SNAP Connect* 60
ID 19 – Unicast Retries 60
ID 20 – Mesh Routing Maximum Timeout 60
ID 21 – Mesh Routing Minimum Timeout 60
ID 22 – Mesh Routing New Timeout 61
ID 23 – Mesh Routing Used Timeout 61
ID 24 – Mesh Routing Delete Timeout 61
ID 25 – Mesh Routing RREQ Retries 61
ID 26 – Mesh Routing RREQ Wait Time 61
ID 27 – Mesh Routing Initial Hop Limit 61
ID 28 – Mesh Routing Maximum Hop Limit 62
ID 29 – Mesh Sequence Number 62

ID 30 – Mesh Override 62
ID 31 – Mesh Routing LQ Threshold, not used by *SNAP Connect* 62
ID 32 – Mesh Rejection LQ Threshold, not used by *SNAP Connect* 62
ID 33 – Noise Floor, not used by *SNAP Connect* 62
ID 34-38 – Reserved for Future Use, not used by *SNAP Connect* 63
ID 39 – Radio LQ Threshold, not used by *SNAP Connect* 63
ID 40 – SNAPpy CRC, not used by *SNAP Connect* 63
ID 41 – Platform, not used by *SNAP Connect* 63
ID 42-49 – Reserved for Future Use 63
ID 50 – Enable Encryption 63
ID 51 – Encryption Key 63
ID 52 – Lockdown 64
ID 53 – Maximum Loyalty, not used by *SNAP Connect* 64
ID 54 – Reserved for Future Use 64
ID 55 – Alternate Encryption Key (*SNAP Connect* only) 64
ID 56-59 – Reserved for Future Use 64
ID 60 – Last Version Booted *(Deprecated)*, not used by *SNAP Connect* 64
ID 61 – Reboots Remaining, not used by *SNAP Connect* 64
ID 62 – Reserved for Future Use 64
ID 63 – Alternate Radio Trim value, not used by *SNAP Connect* 64
ID 64 – Vendor-Specific Settings, not used by *SNAP Connect* 64
ID 65 – Clock Regulator, not used by *SNAP Connect* 65
ID 66 – Radio Calibration Data, not used by *SNAP Connect* 65
ID 67-69 – Reserved for Future Use 65
ID 70 – Transmit Power Limit, not used by *SNAP Connect* 65
ID 71-127 – Reserved for Future Use 65
ID 128-254 – Available for User Definition 65
ID 255 – Reserved for Synapse Use, not used by *SNAP Connect* 65

8. *SNAP Connect* API Reference – Exceptions 66

9. *SNAP Connect* API Reference – HOOKS 72

HOOK_SNAPCOM_OPENED – SNAP COMmunications established 72
HOOK_SNAPCOM_CLOSED – SNAP COMmunications ended 72
HOOK_SERIAL_OPEN – Serial Communications started 73
HOOK_SERIAL_CLOSE – Serial Communications ended 73

HOOK_RPC_SENT – RPC packet processing complete 74
HOOK_STDIN – Data Mode data received 75
HOOK_TRACEROUTE – Trace Route data received 75
HOOK_TOPOLOGY_POLL_RESULTS – Topology poll results available 76
HOOK_OTA_UPGRADE_COMPLETE – Over the air upgrade complete 77
HOOK_OTA_UPGRADE_STATUS – Over the air upgrade status report 77
10. *SNAP Connect* API Reference – Authentication Realms 78
11. *SNAP Connect* API Reference – Sniffer Descriptors 79
snap.DataModeDescriptor – Data Mode Packets 79
snap.MeshDescriptor – Mesh Routing Packets 79
snap.RawDescriptor – Undecoded Packets 80
snap.RpcDescriptor – RPC Packets 80
snap.TraceRouteDescriptor – Trace Route Packets 80

1. Overview

SNAP® Connect enables you to write applications for PCs and embedded gateways that participate in SNAP networks. SNAP, an IoT network platform from Synapse, provides seamless remote control and monitoring capability across distributed networks, including the ability to develop and deploy new application software on remote SNAP instances, or "SNAP nodes." The nodes within a typical SNAP network range from tiny embedded devices with 8-bit processors to embedded Linux boxes, desktop PCs, and even cloud-servers. All types of SNAP nodes are programmed in the Python language, which combines ease of use with portable execution on a virtual machine architecture.

All SNAP Nodes:

- **Route SNAP traffic** through the network using available platform interfaces (TCP/IP, serial, USB)
- **Execute Python scripts**, providing full programmer access to the host platform's capabilities
- **Have a unique SNAP network address**, so they can be found by other SNAP nodes

SNAP Connect is an implementation of SNAP for your PC. It can be hosted on full Python-capable platforms such as Windows, Linux, and OSX.

Why run SNAP on a PC?

Typical applications are:

User Interface

- Manufacturing programming and test stations
- Command-line
- Standalone Windows GUI
- Web Server back-end

Data Logger

- Collect remote data and store it in a database or file-system

Interface Adaptor

- MODBUS TCP server, fronting for remote wireless devices
- Legacy application protocols (serial or TCP/IP)

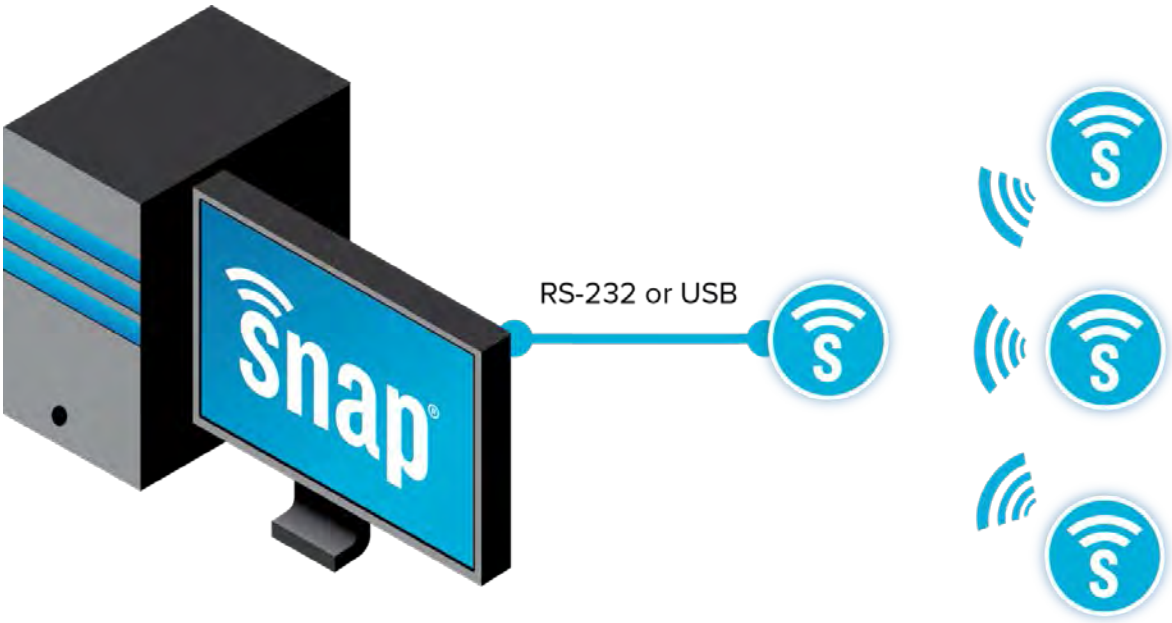
Mesh Aggregator / Internet Bridge

- Unify remote device networks into one network
- Access remote networks from the Internet

SNAP Connect can be used as a gateway "right out of the box," or it can be integrated into software applications that you develop in Python. Embedded SNAP nodes can be used to route SNAP mesh traffic even when no user application script is loaded. With SNAP Connect, this capability is present in the form of a basic example application (Python script) included in the optional *SNAP Connect* examples. Look for `simpleSnapRouter.py`, documented later in this manual. This standalone SNAP instance listens for connections over TCP/IP, while command line options support opening serial and/or USB connections to supported SNAP radio bridge devices.

Introducing SNAP Connect

SNAP Connect is a pure Python package to interface your application to a SNAP network. This Python package is a full SNAP implementation, allowing you to create your own programs that natively interact with the SNAP network.



If you are not already familiar with SNAP networking and SNAPpy scripting please refer to Synapse's **SNAP Primer**, **SNAP User Guide**, and **SNAP Reference Manual** available from http://www.synapse-wireless.com. (These documents are also available within the Help menu of the Synapse Wireless *Portal* software.) It is strongly recommended that you become familiar with the contents of the SNAP Reference manual and have the *Portal* software installed before attempting to develop custom applications using *SNAP Connect*. This document also assumes that you have a good working knowledge of the Python programming language.

How does my PC know what SNAP Address it should use?

Each instance of SNAP Connect requires a license file. This license file specifies what SNAP addresses you are authorized to use.

If you are running SNAP Connect on a Synapse SNAP Connect Appliance (E10/E15/E20):

You already have a full license – just skip to the next section of this document.

If you are running SNAP Connect on a desktop PC:

SNAP Connect comes with a default `License.dat` file that assigns SNAP network Address 00.00.20 to your PC. (This is similar to how Portal defaults to address 00.00.01).

If you need your PC to have a different SNAP Address (for example, you are deploying multiple PCs in the same SNAP network), you will need to purchase an alternate `License.dat` file.

Contact Synapse at 1-877-982-7888 or e-mail support@synapse-wireless.com to order one or more SNAP Connect licenses.

Once you have obtained your replacement `License.dat` file, copy it to the same directory as your application software.

SNAP Connect Licensing

There are several licensing options available when using *SNAP Connect*:

Option 1 – Free License (10 nodes or less)

This option for *SNAP Connect* was introduced to make it easier for customers to quickly get started working on their SNAP applications. Previously, evaluators had to request a time-limited *SNAP Connect* evaluation license file, and could not run *SNAP Connect* at all until they had done so. Now a "fixed address, 10-node" license is bundled in with the *SNAP Connect* library.

Applications that only require a small number of SNAP nodes (up to 10 embedded SNAP Nodes, not counting the *SNAP Connect* application and optionally *Portal*) require no additional license.

Note: Every node on a SNAP network must have a unique SNAP Address. The address of the *SNAP Connect* node will be fixed at SNAP Address 00.00.20 unless you explicitly acquire one or more additional licenses or are running on a SNAP Connect Appliance. This is separate to the address assigned to the RF node that serves as the physical bridge into the RF network.

To move your *SNAP Connect* instance to a different (unique) address, you will have to license an additional SNAP MAC Address. Contact Synapse customer support for pricing and availability. This will be sent to you in the form of a replacement `License.dat` file.

Option 2 - Unrestricted License (more than 10 nodes)

Contact Synapse for pricing and availability.

Applications that require larger SNAP networks are still required to obtain a full *SNAP Connect* license, just as with previous versions.

Note that the unrestricted license will include a replacement SNAP Address; you will no longer be restricted to SNAP Address 00.00.20.

2. Installation

NOTE – If you are using a SNAP Connect E10, it comes with SNAP Connect preinstalled. You can skip ahead to the next section. If you are upgrading the Linux system on an E10, you will need to reinstall *SNAP Connect* after you reinstall the Linux image.

SNAP Connect requires a copy of Python version 2.6 or 2.7 installed on the target computer. (*SNAP Connect* does not currently support Python version 3.x.)

If you don't already have Python installed on your development computer, it can be downloaded from http://www.python.org.

Additionally, you'll need "pip", the recommended 3rd party tool for installing Python packages. You can find instructions for installing pip on your system at https://pip.pypa.io/en/stable/installing/.

If AES-128 support is required, the third-party library PyCrypto (http://pycrypto.org/) is also needed. If you do not know whether you will need AES-128 support, you can go ahead and install *SNAP Connect* now and install the PyCrypto library later when it is needed.

Install via pip

```
pip install -i https://update.synapse-wireless.com/pypi snapconnect
```

(Optional) Cache the download to transfer via USB key

Run the following as a single command:

```
pip install --download="/mypathtosave/"
            –i https://update.synapse-wireless.com/pypi snapconnect
```

The files necessary to install SNAP Connect will be in the path you specify.

To install from the cache:

```
pip install --no-index --find-links="/mypathtosave/" snapconnect
```

(Optional) – Unzip the examples to your computer

Several example *SNAP Connect* applications are bundled into an `examples.zip` file, which is available from https://forums.synapse-wireless.com/showthread.php?t=9.

Using the tool of your choice (7ZIP, PKZIP, etc.), unzip the file into a working directory on your computer.

NOTE – Be sure to specify "use directory paths" when you unzip the examples!

This will create an "examples" directory tree with a "manual" subdirectory (plus several others) underneath it.

As you might guess, the "manual" subdirectory contains all the examples from this User Manual.

You will also see subdirectories such as:

DisplayInfo

EchoTest

SpyUpload

Etc.

Each contains another example program.

For more about the available example programs, refer to section Additional Examples within this document.

Sample Application: SimpleMcastCounter

If you have previously used SNAP then you are probably familiar with the multi-cast counter example. In this example we will show how to write an equivalent Python application using *SNAP Connect*, based on the SNAPpy `McastCounter.py` script. Let's take a look at the contents of our sample application:

```
import logging
from snapconnect import snap

class McastCounterClient(object):
    def __init__(self):
        # Create a SNAP instance
        self.snap = snap.Snap(funcs={'setButtonCount': self.set_button_count})
        # Open COM1 (port 0) connected to a serial SNAP bridge
        self.snap.open_serial(snap.SERIAL_TYPE_RS232, 0)
        # Create a logger
        self.log = logging.getLogger("McastCounterClient")

    def set_button_count(self, count):
        # This function is called by remote SNAP nodes
        # Log the current count received
        self.log.info("The button count = %i" % (count))

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO) # print logging messages to STDOUT
    client = McastCounterClient() # Instantiate a client instance
    client.snap.loop() # Loops waiting for SNAP messages
```

Details on all of the parameters that *SNAP Connect* methods take are described in the *SNAP Connect* API section of this manual.

Let's walk through this simple multi-cast counter example to see how it works. The only two packages we import in this example are the standard python logging package and the *SNAP Connect* package. Once we define the McastCounterClient class, we instantiate the *SNAP Connect* instance we will use in the init function. (Look for `self.snap` in the sample code).

After we instantiate the *SNAP Connect* instance we pass it a Python dictionary with the functions we would like to make publicly available to be called by other SNAP nodes. In this example the only publicly available function we register is the function named "setButtonCount", and when the function is called by other SNAP nodes it will execute our "set_button_count" method. Next we call the method "open_serial" on our *SNAP Connect* instance to have it open a serial connection to the SNAP bridge node. Finally, we create a logging instance to log anything we receive from a SNAP node.

The "set_button_count" method accepts one parameter, which contains the current count as sent by the remote SNAP node. When this method is called our function uses the saved logger instance and logs the current button count.

Within the main section we first configure the default logger to print any logged messages to STDOUT. We then create an instance of the client class that we defined earlier in the example code. At the end we call the loop method of the *SNAP Connect* instance to start having *SNAP Connect* continuously send, receive, and process SNAP messages. Until we call the loop method, *SNAP Connect* will not be able to do anything we ask it to do.

To run this program, you should first change the open_serial() COM port number to that of your bridge device. If you're using a USB SnapStick, you'll need to change the SERIAL_TYPE to match also (see the open_serial API documentation later in this manual). Use Portal to verify the bridge device is working, and to upload the standard "McastCounter" example (installed with Portal) to a wireless test-device (e.g. a Synapse SN171 protoboard with a SNAP Engine installed).

Now disconnect Portal from the bridge, since only one program at a time can have the serial connection open, and your SNAP Connect program will be connecting. At this point, you will be able to run the Python program:

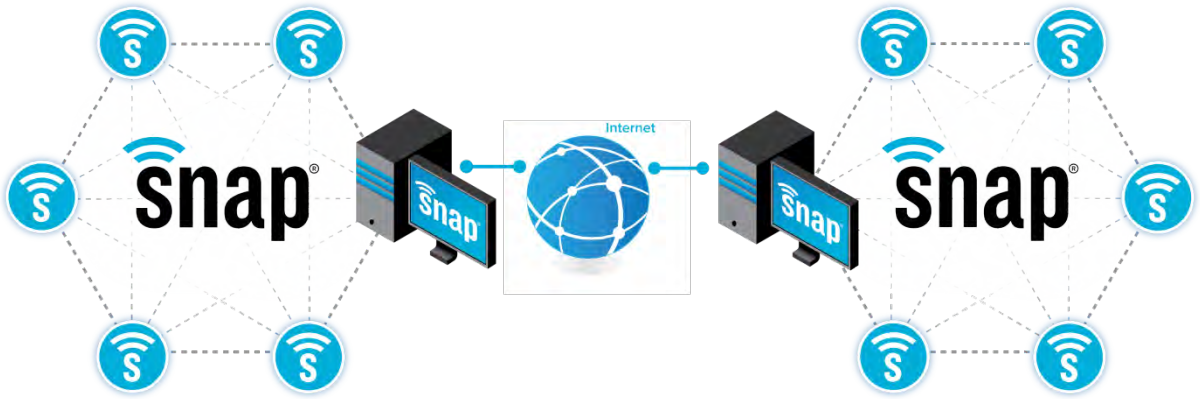
```
>python SimpleMcastCounter.py

Permanent license created on 2015-09-15 23:01:39.808000 for 000020
```

If all went well, you will see a message similar to the above message, and the program will be waiting for SNAP messages. Now you can press the button on your wireless test device and the "button count" log messages will appear on your computer console. After testing this program, press **Ctrl-C** to terminate the test.

Remote Connection Capabilities

You can connect multiple *SNAP Connect* instances together over a local TCP/IP network or the Internet. This capability allows SNAP networks to route their messages over radio interfaces as well as over existing TCP/IP networks.

For example, let's say you have an existing SNAP network located in Huntsville, Alabama. Recently you opened a new location with a SNAP network in San Jose, California, and you would like to connect the two SNAP networks together. In your Huntsville location you would configure your *SNAP Connect* application to listen for remote connections. Then, in your San Jose location, you would configure your *SNAP Connect* application to connect to the IP address of the *SNAP Connect* application in Huntsville. This would enable SNAP messages from each location to be routed over the Internet, allowing for SNAP nodes from each location to communicate with each other.



In general there are no configuration changes that need to be made on your existing SNAP nodes to enable this capability. However, it is important to note that each *SNAP Connect* instance counts as a mesh routing hop. Be sure to open the appropriate port on your firewall to allow traffic to flow. Depending on how your SNAP nodes are configured the maximum number of hops mesh routing messages can take may need to be increased. Also, depending on how much latency there is on the TCP/IP connection between the two *SNAP Connect* instances, the mesh routing timeouts may need to be adjusted.

Remote SNAP TCP/IP connections require authentication between the two ends using digest access authentication over the network to protect login credentials. There are default authentication credentials (username = "public" and password = "public") that are used if none are provided.

If you prefer, you can create your own system for supplying and validating credentials by adding one function on the client end to provide the username and password, and another function on the server end to validate that the specified username/password pair is correct. These functions can use hard-coded pairs, database or file lookups, or user prompts to generate their data, according to your needs. To replace the default authentication, specify the names of your new routines in your connect_tcp() function call on your client, and accept_tcp() function on your server. (See the API section for more details on these functions.)

NOTE – SNAP Connect only supports IPv4 addresses for TCP connections.

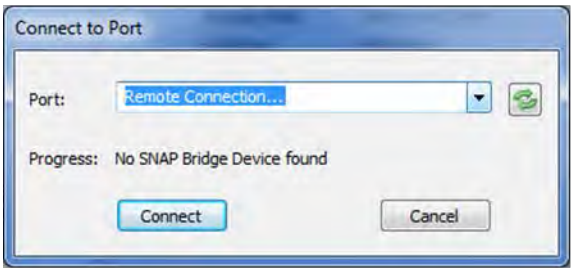
Sample Application: Remote TCP Access

In this sample application we will be building on our previous multi-cast counter example, by showing how to enable remote TCP connections. SNAP Connect programs can connect to each other over an IP network, forwarding routing and RPC traffic among all connected interfaces. A common use of this capability is to allow a remote TCP/IP connection from Portal. A small modification of our previous example allows it to accept connections from other SNAP Connect instances, including Portal.

```
import logging
from snapconnect import snap

class McastCounterClient(object):
    def __init__(self):
        # Create a SNAP instance
        self.snap = snap.Snap(funcs={'setButtonCount': self.set_button_count})
        # Open COM1 (port 0) connected to a serial SNAP bridge
        self.snap.open_serial(snap.SERIAL_TYPE_RS232, 0)
        # Accept connections from other SNAP Connect instances and Portal
        self.snap.accept_tcp()

        # Create a logger
        self.log = logging.getLogger("McastCounterClient")

    def set_button_count(self, count):
        # This function is called by remote SNAP nodes
        # Log the current count received
        self.log.info("The button count = %i" % (count))

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO) # print logging messages to STDOUT
    client = McastCounterClient() # Instantiate a client instance
    client.snap.loop() # Loops waiting for SNAP messages
```

As you have probably noticed, this script is almost exactly the same as our first sample application. The main difference here is that in addition to connecting to a SNAP bridge node via serial port, we configure the SNAP instance to listen for and accept remote SNAP TCP/IP connections.

When you run this program, you can test it with your remote SNAP device as with the first example and see the button count logged to the application console. But now you can simultaneously connect to this network with *Portal*. In *Portal*'s connect dialog, select "Remote Connection…" and then connect to *localhost* (or 127.0.0.1).



With Portal connected over TCP/IP you can do a broadcast-ping and see all SNAP nodes in your network, including the SNAP Connect example program above, as well as its bridge node and remote node(s). From Portal's node-info panel, invoke the 'incrementCount()' function on the remote SNAP device. You will see the count LEDs increment on the remote device itself, and the SNAP Connect console will log the event. You can also invoke a multicast call to the 'incrementCount()' function from *Portal*'s command line window and get the same result.

Recommended Practices

When writing your *SNAP Connect*-based application there are a few recommended practices.

Hooks

The first recommend practice is to take advantage of the available hooks for controlling which code is executed when events occur in *SNAP Connect*. The available hooks are listed in the *SNAP Connect* API section, but typically the most important to remember are HOOK_RPC_SENT and HOOK_SNAPCOM_OPENED.

Just as HOOK_RPC_SENT is important in a SNAPpy script, it is important in *SNAP Connect* applications. This hook allows your program to know when the system has completed processing a specific outgoing RPC (either unicast, multicast, or directed multicast), and if it was queued for transmission successfully. Based on this knowledge it might be appropriate for your application to send another RPC or know that there is room in the queue to send another RPC. Just as in SNAPpy, the rpc() method (as well as the mcastRpc() and dmcastRpc() methods) returns immediately and does not mean your RPC has been, or even will be, sent. This hook does not trigger until *SNAP Connect* is done processing the specified packet (whether that processing was successful or not).

HOOK_SNAPCOM_OPENED informs your application that you have successfully connected to a remote *SNAP Connect* instance or another *SNAP Connect* instance has connected to you. This is important to know if your application depends on having connectivity to another instance. If your application is not connected to another *SNAP Connect* instance or a SNAP bridge node, all of your RPCs will fail because there is no one else to talk to.

Note – The timing hooks used in SNAPpy scripts are not available in SNAP Connect applications. Instead you can using the timing mechanisms in Python, such as the scheduler in the sched module.

Asynchronous Programming

SNAP Connect is designed to run in a single process within a single thread. As you have seen in our sample applications, once *SNAP Connect* and your application are initialized, we call the *SNAP Connect* loop() method. This method is a convenience method that is a "while True" loop that continually calls the *SNAP Connect* poll() method:

```
def loop(self):
    while True:
        self.poll()
        time.sleep(0.001)
```

As mentioned in the *SNAP Connect* API, the poll method calls the necessary components that *SNAP Connect* uses. Since your program is essentially always running that loop, your application must be programmed in an asynchronous fashion.

To aid in writing asynchronous programs, access to the asynchronous event scheduler that *SNAP Connect* uses is available.

Scheduling Asynchronous Events

The event scheduler is straightforward to use and allows your program to schedule functions to be called. Access to the event scheduler is available by calling methods on the "scheduler" property of your *SNAP Connect* instance. To schedule a function to be called later you would call the schedule method. For example, to schedule sending a multi-cast RPC call every one and a half seconds you could enter:

```
def my_func(self):
    self.snap.mcast_rpc(1, 1, "hello")
    return True
def start_mcasting(self):
    self.snap.scheduler.schedule(1.5, self.my_func)
```

The signature for the schedule function looks like:

```
def schedule(self, delay, callable, *args, **kwargs)
```

where the parameters are:

`delay` : The delay, in seconds, to use before calling the callable argument. A delay of 0 indicates the callable should be called on the next poll interval

`callable` : The callable object to schedule

`*args, **kwargs` : Optional arguments and keyword argument to pass to the callable object

The schedule function returns an EventElement object that has a method called "Stop" that enables you to cancel the scheduled event (in other words, cancel having the callable be called).

For example, to schedule a timeout that can be disabled by calling a specific function you could do:

```
def start_timeout(self):
    self.my_timeout = self.snap.scheduler.schedule(10.0, self.on_timeout)

def disable_timeout(self):
    self.my_timeout.Stop()
```

The last thing to know about using the event scheduler is how to re-schedule a running function after it runs. When your function returns after being called by the event scheduler, its return value is checked. If the return value is `True`, then the event is automatically rescheduled using the original delay value. If you would like to reschedule with a different delay, your function can return a number (integer or float, indicating a delay duration in seconds) as a different valid delay value. If you don't want your function to be rescheduled at all, your function can return `None` or `False`.

```
def reschedule_every_second():
      print "Tick"
      return 1
comm.scheduler.schedule(0, reschedule_every_second)

def reschedule_for_original_delay():
      print "Tick"
      return True #This will be rescheduled every 5 seconds in this example
comm.scheduler.schedule(5, reschedule_for_original_delay)
```

```
def do_not_reschedule():
      print “Tick”
      return False
comm.scheduler.schedule(5, do_not_reschedule)
```

Be careful when scheduling *SNAP Connect* functions that have return values. For example, the command

```
comm.scheduler.schedule(0, comm.rpc, addr, ‘foo’)  # <-- DON’T DO THIS!
```

looks like it would schedule a single RPC call. However, the `rpc` function actually returns an integer for identifying the packet. This integer will cause the `rpc` function to be continually rescheduled. Instead, you can use a wrapper that will make the call and return `None` to avoid rescheduling.

```
def rpc_with_no_return(*args):
      comm.rpc(*args)
      return None
comm.scheduler.schedule(0, rpc_with_no_return, addr, ‘foo’)
```

Multiple Instances

When attempting to run multiple instances of *SNAP Connect* it is important that each instance uses a unique SNAP address. Just as regular SNAP nodes need a unique address to differentiate each other over the air, *SNAP Connect* instances need unique addresses on the network as well. A single license file can contain more than one SNAP network address. When you instantiate your *SNAP Connect* instance you can provide as a keyword argument which SNAP network address out of the license you would like to use. See the __init__() function for more details.

3. Examples: SNAP Connect on the E10 Gateway

Previous examples have been somewhat "generic," since we had no way of knowing exactly what platform was running *SNAP Connect*.

One of the platforms that can run *SNAP Connect* are the Synapse *SNAP* Gateway appliances, such as the E10, E15, and E20..

The *SNAP Connect* E10 is a small Linux computer that comes with *SNAP Connect* preloaded.

In this section, we go over some example code that is specific to the E10. The intent is that by showing something concrete, it will further clarify the material covered previously in this document.

For the purpose of these examples, all you need to know about the E10 is that it has both an internal SNAP Engine *and* a separate ARM processor that runs the Linux OS (on which the *SNAP Connect* application is running). The SNAP Engine connects to the ARM processor using its serial ports.

The SNAP Engine has control of the tricolor LED labeled "A," and the Linux processor has control of the tricolor LED labeled "B" as well as direct access to the Push Button labeled "MODE." (Tricolor LEDs can display green or red, or amber as a combination of green and red.)

The E10 runs a *SNAP Connect* application (UserMain.py, by default) when it powers up. In order to run these examples you must stop any existing *SNAP Connect* application before starting a new one.[1]

```
> sudo /etc/init.d/S999snap stop
Stopping SNAP Connect...OK
>
```

You can remove (or modify) the "S999snap" startup script and add your own SNAPConnect application to run at power up.

For more about the E10, refer to the "*SNAP Connect* E10 User Guide" available from www.synapse-wireless.com.

UserMain.py

The following is the example Python script included in the manual directory of the optional *SNAP Connect* examples. An E10 may come preloaded with a version of this script that provides additional capabilities.

First we list the entire file, then we list it again with inline commentary.

[1] The 1.1 version of the E10 operating system requires the use of the sudo prefix before the command, as shown. The 1.0 version of the E10 operating system allows you to connect as root, and does not require the sudo prefix.

```
#
# A minimal example SNAP Connect 3.4 application. This example was
# deliberately kept simple, and so does not showcase the full power
# of Python or SNAP Connect 3.4. It enables RPC access via TCP/IP
# or via SNAP Engine (2.4 GHz radio, etc.), and gives you control
# of the Linux processor's "B" LED and Push Button.
#
#
# Note: This example is intended to work with E10 Version 1.1
# If you are using an old version of the E10, see the notes in
# setGreenLed, setRedLed, and readButton for necessary changes.

from snapconnect import snap
import os  # So we can gain access to the LEDs and Push Button
import time  # So we can sleep in our loop

#
#
# Example routines to control the "B" LED on the E10
# (The "A" LED is connected to the internal SNAP Engine inside the E10)
#
#

#
# Individual LED pin control (used internally)
#
def setGreenLed(value):
    #Note: For E10 1.0 use the following instead:
    #os.system("echo "+str(value)+" >> /sys/class/leds/greenled/brightness")
    os.system("greenled "+str(value))

def setRedLed(value):
    #Note: For E10 1.0 use the following instead:
    #os.system("echo "+str(value)+" >> /sys/class/leds/redled/brightness")
    os.system("redled "+str(value))

#
# It's actually a tri-color LED (this will be the public API)
#
def setLedBOff():
    setGreenLed(0)
    setRedLed(0)

def setLedBGreen():
    setGreenLed(1)
    setRedLed(0)

def setLedBRed():
    setGreenLed(0)
    setRedLed(1)

def setLedBYellow():
    setGreenLed(1)
    setRedLed(1)

#
# Example routine to read the MODE push button
# Normally a pullup resistor holds the pin high/True
# Pushing the button connects the physical pin to ground (low/False)
#
```

```
def readButton():
    #NOTE: For the E10 1.0 use the following instead
    # result = os.system("/usr/bin/gpio9260 ?PB10")
    result = os.system("/usr/bin/button")
    return result != 0 # Forcing boolean result

def server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm
    username : The username specified by the remote server
    """
    if username == "public":
        return "public"


if __name__ == '__main__':
    import logging

    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s:%(msecs)03d
%(levelname)-8s %(name)-8s %(message)s', datefmt='%H:%M:%S')

    funcdir = { # You CHOOSE what you want to provide RPC access to...
                'setLedBOff'    : setLedBOff,
                'setLedBGreen'  : setLedBGreen,
                'setLedBRed'    : setLedBRed,
                'setLedBYellow' : setLedBYellow,
                'readButton'    : readButton
              }
    com = snap.Snap(funcs=funcdir)

    # Make us accessible over TCP/IP
    com.accept_tcp(server_auth)

    # Make us accessible over our internal SNAP Engine
    com.open_serial(1, '/dev/ttyS1')

    #
    # Configure some example settings
    #

    # No encryption
    com.save_nv_param(snap.NV_AES128_ENABLE_ID, False)

    # Lock down our routes (we are a stationary device)
    com.save_nv_param(snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)

    # Don't allow others to change our NV Parameters
    com.save_nv_param(snap.NV_LOCKDOWN_FLAGS_ID, 0x2)

    # Run SNAP Connect until shutdown
    while True:
        com.poll()
        time.sleep(0.001)
    com.stop_accepting_tcp()
```

Let's go through this file again, section by section.

```
#
# A minimal example SNAP Connect 3.4 application. This example was
# deliberately kept simple, and so does not showcase the full power
# of Python or SNAP Connect 3.4. It enables RPC access via TCP/IP
# or via SNAP Engine (2.4 GHz radio, etc.), and gives you control
# of the Linux processor's "B" LED and Push Button.
#

from snapconnect import snap
```

The above import provides access to the *SNAP Connect* library. The "snapconnect" in the above statement refers to the directory containing that library. If your directory has a different name, modify the import statement appropriately. (For example, early versions of the E10 had this package in the Snap directory instead of the snapconnect directory.)

```
import os  # So we can gain access to the LEDs and Push Button
import time  # So we can sleep in our loop
```

The "os" and "time" modules are part of the standard Python library of routines that allow you to do things like invoke other programs. We are taking advantage of the fact that we already know how to control the LED and read the button from the Linux command line, and will use OS calls to those command-line commands from within our script. (Refer to the E10 User Guide.)

We also wanted to showcase the idea that if your platform can do it, you can access that functionality from *SNAP Connect*. For features that are directly supported by Python, just import the corresponding module. For features that do not have direct Python support, you can access the Linux command line.

```
#
#
# Example routines to control the "B" LED on the E10
# (The "A" LED is connected to the internal SNAP Engine inside the E10)
#
#


#
# Individual LED pin control (used internally)
#
def setGreenLed(value):
    #Note: For E10 1.0 use the following instead:
    #os.system("echo "+str(value)+" >> /sys/class/leds/greenled/brightness")
    os.system("greenled "+str(value))

def setRedLed(value):
    #Note: For E10 1.0 use the following instead:
    #os.system("echo "+str(value)+" >> /sys/class/leds/redled/brightness")
    os.system("redled "+str(value))
```

The "system()" function in the standard Python "os" module allows you to invoke any function you could do directly from the command line.

On the E10's 1.1 version of Linux, commands like…

```
greenled 0
redled 1
```

…can be used to write directly to the LED hardware.

By wrapping these two primitive operations up in subroutines, we hide the complexity from the rest of the code.

```
#
# It's actually a tri-color LED (this will be the public API)
#

def setLedBOff():
    setGreenLed(0)
    setRedLed(0)

def setLedBGreen():
    setGreenLed(1)
    setRedLed(0)

def setLedBRed():
    setGreenLed(0)
    setRedLed(1)

def setLedBYellow():
    setGreenLed(1)
    setRedLed(1)
```

Above you can see that we use those primitive access routines to implement a nicer API. Note that just because the routines are implemented, that does not mean they can be called via RPC. We specify that *later* in this same code.

```
#
# Example routine to read the MODE push button
# Normally a pullup resistor holds the pin high/True
# Pushing the button connects the physical pin to ground (low/False)
#
def readButton():
    #NOTE: For the E10 1.0 use the following instead
    # result = os.system("/usr/bin/gpio9260 ?PB10")
    result = os.system("/usr/bin/button")
    return result != 0 # Forcing boolean result
```

Here we again use the ability to run command line programs – in this case, the “button” program that comes preloaded on the E10, to read the MODE button on the front of the E10.

```
def server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm
    username : The username specified by the remote server
    """
    if username == "public":
        return "public"
```

Here we provide an explicit “authorization” routine for the E10. By changing this routine, more complex authorization mechanisms could be implemented. For example, users and passwords might get looked up in a database, or at a minimum there might be separate hard-coded username/password pairs per realm.

```
if __name__ == '__main__':
```

```
    import logging

     logging.basicConfig(level=logging.DEBUG, format='%(asctime)s:%(msecs)03d
%(levelname)-8s %(name)-8s %(message)s', datefmt='%H:%M:%S')
```

Here we are using standard Python logging. Nothing about the above three lines of code is specific to *SNAP Connect*.

```
    funcdir = { # You CHOOSE what you want to provide RPC access to...
                'setLedBOff'    : setLedBOff,
                'setLedBGreen'  : setLedBGreen,
                'setLedBRed'    : setLedBRed,
                'setLedBYellow' : setLedBYellow,
                'readButton'    : readButton
              }
    com = snap.Snap(funcs=funcdir)
```

We mentioned previously that just because a function is defined does not automatically make it callable from other SNAP Nodes. Here the code fills in a Python dictionary (notice the {…}) with the set of Python functions that are to be callable via RPC, as well as the names we want to use. The code enables the four functions to set the LEDs to off or one of three colors, but does not include the more fundamental functions that specifically control the red and green LEDs.

In this example the mapping is direct and one-to-one (nothing was renamed), but the renaming capability is there when you need it. That allows you to maintain a constant set of names for invoking your functions even if you change the names of the functions themselves. It also allows you to create aliases for your functions, much as *SNAP Connect* has both mcastRpc() and mcast_rpc() callables in its API for the same function.

```
    # Make us accessible over TCP/IP
    com.accept_tcp(server_auth)
```

We tell *SNAP Connect* to listen for incoming TCP/IP connections, so *Portal* or another *SNAP Connect* instance can connect to us.

```
    # Make us accessible over our internal SNAP Engine
    com.open_serial(1, '/dev/ttyS1')
```

There is not much mysterious here, once you know that “/dev/ttys0” is the command line console on the E10, and “/dev/ttyS1” is the UART that connects to the serial port on the internal SNAP Engine.

```
    #
    # Configure some example settings
    #

    # No encryption
    com.save_nv_param(snap.NV_AES128_ENABLE_ID, False)

    # Lock down our routes (we are a stationary device)
    com.save_nv_param(snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)

    # Don't allow others to change our NV Parameters
    com.save_nv_param(snap.NV_LOCKDOWN_FLAGS_ID, 0x2)
```

If we did want to enable encryption, we would need to also specify the 16-byte encryption key in the snap.NV_AES128_KEY_ID parameter.

Setting the “max timeout” for mesh routing to 0 reduces the number of “route discoveries” that have to take place. If your network is not fixed and stable, setting this to 0 may not be a good idea.

Since we consider the E10 in our example to be the master, we want to maintain local control over its NV parameters. The E10 allows a value of 2 for the lockdown parameter to prevent changes to any NV parameters. (This is not a valid value in most SNAP nodes, where a value of 1 prevents over-the-air changes to the SNAPpy script in the node.)

Other settings for these parameters (as well as many other parameters) are possible. The above is only one example configuration.

```
# Run SNAP Connect until shutdown
while True:
    com.poll()
    time.sleep(0.001)
com.stop_accepting_tcp()
```

We could have just called the `com.loop()` function and done away with the while loop. However, we wanted to show that your Python program can remain active, even with the *SNAP Connect* library active.

This particular example program is essentially “passive,” in that it waits for other nodes to call on it (via RPC).

A more “active” program could perform additional functions after every call to com.poll(). For example, it could monitor the MODE pushbutton (we’ve already provided a function to do the actual “read”), and could invoke an RPC (or take some other action) whenever that button was pressed. As it stands, we’ve enabled monitoring of the button but perform no such monitoring locally.

With this script running on the E10, any node networked to the E10 can send an RPC to the E10 to control the device’s LED color. That function call can originate from the SNAP Engine contained within the E10, a SNAP node connected to that SNAP Engine over the air, or a SNAP node networked to the E10 over the TCP/IP connection.

Sample Application – simpleSnapRouter.py

The following is one of the example Python scripts that are included with *SNAP Connect*. (Look in the “examples\manual” subdirectory.) The simpleSnapRouter.py script performs exactly that function: it is a simple SNAP router. A PC or device running this script with *SNAP Connect* forms a nexus between any of several disparate networks.

For example, in addition to a TCP/IP connection, you could make a connection over a COM port to a network of 900 MHz SNAP nodes, make a connection over an SS200 to a network of 2.4 GHz SNAP nodes running at 2 Mbps, and make a connection over the USB port to a network of 2.4 GHz SNAP nodes running at 250 Kbps – and all three of these subnetworks would be able to communicate with each other through the *SNAP Connect* node.

Usage

Here are some examples of launching simpleSnapRouter.py:

```
python simpleSnapRouter.py –-help
```

Launching the application this way will display info on the various command line arguments, and then the program will exit. Note that there are two hyphens before the word “help”. The following command performs the same function.

```
python simpleSnapRouter.py –h
```

You will notice a pattern here. For most of the command line arguments, there is both a long form (a keyword) prefixed by two hyphens, and a short form (single character) prefixed by one hyphen. So for example, --help and -h are equivalent.

```
python simpleSnapRouter.py
```

Running the application with no command line arguments will result in its default behavior – it will listen for incoming TCP/IP connections, but will not enable any other interfaces (even if they are present).

```
python simpleSnapRouter.py –-deaf
```

or

```
python simpleSnapRouter.py –d
```

Running the application with this command line argument (either form) will override the default "listening" behavior. Incoming TCP/IP connections will be rejected.

NOTE – Using this option by itself is not very useful. *SNAP Connect* will not make any connection by itself, and will not accept any connection from any other node.

```
python simpleSnapRouter.py <IP Address>
python simpleSnapRouter.py 192.168.1.55 192.168.1.56
```

Running the application with one or more valid IP addresses as command line arguments will force outbound connections to be attempted to other *SNAP Connect* instances. This includes other instances of simpleSnapRouter.py.

```
python simpleSnapRouter.py –-com <COM Port Number>
python simpleSnapRouter.py –c <COM Port Number>
python simpleSnapRouter.py –c 0
```

By specifying a "COM" port using the -c or --com options (0 = COM1, 1=COM2, etc.) you can enable the SNAP Packet Serial protocol on the specified serial port. Assuming there is a live bridge node connected and listening on that COM port, this will allow simpleSnapRouter.py to communicate over the air to other SNAP Nodes.

NOTE – COM ports include both "real" RS-232 ports as well as many common brands of "USB Serial" adapter. Some software packages establish virtual COM ports as communication interfaces, too. As long as the interface looks like a COM port, you can connect to it this way.

```
python simpleSnapRouter.py –-ss200 <SS200 Number>
python simpleSnapRouter.py –s <SS200 Number>
python simpleSnapRouter.py –s 0
```

By specifying a "SS200" device (the first one detected by your computer is number 0, the next one plugged in will become number 1, etc.) you can enable the SNAP Packet Serial protocol on the specified SNAP Stick 200.

```
python simpleSnapRouter.py –-legacy <Synapse USB Device Number>
python simpleSnapRouter.py –l <Synapse USB Device Number>
python simpleSnapRouter.py –l 0
```

By specifying a "legacy" device (the first one detected by your computer is number 0, the next one plugged in will become number 1, etc.) you can enable the SNAP Packet Serial protocol on the specified USB interface. Examples of legacy devices include the SN132 SNAP Stick and the SN163 Bridge Demonstration Board.

Note that as written this application only supports enabling one of each type of serial connection. In other words, `simpleSnapRouter.py` can establish a maximum of three simultaneous serial connections, assuming your computer has a COM port, an SS200, and a legacy USB device connected.

Also note that the example program is written to ignore any "serial" errors, on the assumption that the user would rather the program continue to run with any remaining interfaces still operational. (For example, if the SS200 SNAP Stick were unplugged from the computer, the serial and TCP/IP connections may still be sufficient.)

```
python simpleSnapRouter.py –u <user name>
python simpleSnapRouter.py –-user <username>
python simpleSnapRouter.py –-user public
python simpleSnapRouter.py –p <password>
python simpleSnapRouter.py –-password <password>
python simpleSnapRouter.py –-password public
```

By specifying the --user and --password options, you can override the default credentials used by *SNAP Connect*.

NOTE – If you are going to enable alternate credentials, you need to do so throughout your SNAP Network.

Source Code

First we list the entire `simpleSnapRouter` file. We then provide additional commentary after the source listing.

```
"""

simpleSnapRouter.py - an example for the SNAP Connect User Manual
By default, enables listening for incoming TCP/IP connections.
Command line options allow:
1) NOT listening for incoming TCP/IP connections
2) Establishing a serial connection on a "COM" port
3) Establishing a serial connection on a legacy USB device like a Synapse SN163 or
SN132
4) Establishing a serial connection on a USB device like a SS200
5) Connecting to other SNAP Connect instances over TCP/IP
6) Changing the user, and password settings from their default value of "public"
"""
import logging
from optparse import OptionParser
from snapconnect import snap
DEFAULT_USER = 'public'
DEFAULT_PASSWORD = 'public'
my_user = DEFAULT_USER
my_password = DEFAULT_PASSWORD
def my_client_auth(realm):
    """
    An example client authorization function

    Returns the previously set username and password to the remote server
    to use as it's authorization, *if* the specified realm is correct

    realm : The realm specified by the remote server. For TCP connections
            this will always be snap.REALM_SNAP
    """
    return (my_user, my_password)

def my_server_auth(realm, username):
    """
    An example server authentication function
```

```
    Returns the password for the specified username in the realm

    realm : This server's realm. For TCP connections this will
            always be snap.REALM_SNAP
    username : The username specified by the remote server
    """
    if username == my_user:
        return my_password

def echo(obj):
    """Just a function that can be called remotely"""
    print str(obj)

def main():
    global comm
    global my_user, my_password

    usage = "usage: %prog [options] [other-node-IP-addresses]"
    parser = OptionParser(usage)

    parser.add_option("-d", "--deaf", action="store_true", dest="no_listener",
default=False, help="DON'T listen for incoming TCP/IP connections (default is DO
listen)")

    parser.add_option("-c", "--com", dest="comport", help="open a connection on the
specified COM port (0 = COM1)")
    parser.add_option("-s", "--ss200", dest="ss200", help="open a connection on the
specified SS200 device (0 = SNAPstick0)")
    parser.add_option("-l", "--legacy", dest="usb", help="open a connection on the
specified USB device (0 = USB0)")

    parser.add_option("-u", "--user", dest="user", default=DEFAULT_USER,
help='specify an alternative SNAP Realm user name(default is "public")')
    parser.add_option("-p", "--password", dest="password",
default=DEFAULT_PASSWORD, help='specify an alternative SNAP Realm password (default
is "public")')

    (options, args) = parser.parse_args()
    print options
    print args
        funcdir = {"echo": echo}
    comm = snap.Snap(funcs=funcdir)
    my_user = options.user
    my_password = options.password

    if options.no_listener:
        print "Listening for incoming TCP/IP connections has been disallowed by
user"
    else:
        print "Listening for incoming TCP/IP connections"
        comm.accept_tcp(auth_info=my_server_auth)

    if len(args) != 0:
        # Establish requested outbound TCP/IP connections
        for arg in args:
            print "Establishing a TCP/IP link to "+arg
            comm.connect_tcp(arg, auth_info=my_client_auth)

    # Did the user ask us to open a normal serial port connection (COM1-COMn)?
    if options.comport != None:
        port = int(options.comport)
```

```
        print "Opening a serial connection on COM%d" % (port+1)
        try:
            comm.open_serial(snap.SERIAL_TYPE_RS232, port)
        except Exception, e:
            print "Invalid COM port specified"

    # Did the user ask us to open a connection to a SS200?
    if options.ss200 != None:
        port = int(options.ss200)
        print "Opening a serial connection to SNAPstick%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK200, port)
        except Exception, e:
            print "Invalid SS200 device specified"

    # Did the user ask us to open a connection to a legacy Synapse USB device?
    if options.usb != None:
        port = int(options.usb)
        print "Opening a serial connection on USB%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK100, port)
        except Exception, e:
            print "Invalid USB port specified"

    comm.loop()

if __name__ == '__main__':
    logging.basicConfig(filename='simpleSnapRouter.log', level=logging.DEBUG,
format='%(asctime)s %(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S')
    main()
```

Source Code with Commentary

```
"""
simpleSnapRouter.py - an example for the SNAP Connect User Manual

By default, enables listening for incoming TCP/IP connections.
Command line options allow:
1) NOT listening for incoming TCP/IP connections
2) Establishing a serial connection on a "COM" port
3) Establishing a serial connection on a legacy USB device like a Synapse SN163 or
SN132
4) Establishing a serial connection on a USB device like a SS200
5) Connecting to other SNAP Connect instances over TCP/IP
6) Changing the user, and password settings from their default value of "public"
"""
import logging

from optparse import OptionParser
```

So far, this is pretty standard. The only thing new here is the import of optparse, a standard Python library for working with command line arguments.

```
from snapconnect import snap

DEFAULT_USER = 'public'
DEFAULT_PASSWORD = 'public'

my_user = DEFAULT_USER
my_password = DEFAULT_PASSWORD
```

```
def my_client_auth(realm):
    """
    An example client authorization function

    Returns the previously set username and password to the remote server
    to use as it's authorization, *if* the specified realm is correct

    realm : The realm specified by the remote server. For TCP connections
            this will always be snap.REALM_SNAP
    """
    return (my_user, my_password)

def my_server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm. For TCP connections this will
            always be snap.REALM_SNAP
    username : The username specified by the remote server
    """
    if username == my_user:
        return my_password
```

Here you can see some variables and routines defined such that the default SNAP Authentication Scheme can be overridden. Notice that unless an alternate username and/or password is provided (either by changing the source code or using command line arguments), the resulting behavior is the same.

```
def echo(obj):
    """Just a function that can be called remotely"""
    print str(obj)
```

Here is a classic example of a test function. Once you launch simpleSnapRouter.py, how do you know it is really listening for incoming RPC calls? By invoking the “echo()” function from some other SNAP Node, you can tell from the console output that the RPC exchange took place.

```
def main():
    global comm
    global my_user, my_password
```

Here begins the main() part of this program. When you want to change a global variable from within the scope of a local routine, you have to use global statements to tell Python that is what you intend.

```
    usage = "usage: %prog [options] [other-node-IP-addresses]"
    parser = OptionParser(usage)
```

Here we have created an “Option Parser” and specified what the first line of the “–help” output will be.

```
    parser.add_option("-d", "--deaf", action="store_true", dest="no_listener",
default=False, help="DON'T listen for incoming TCP/IP connections (default is DO
listen)")
    parser.add_option("-c", "--com", dest="comport", help="open a connection on the
specified COM port (0 = COM1)")
    parser.add_option("-s", "--ss200", dest="ss200", help="open a connection on the
specified SS200 device (0 = SNAPstick0)")
```

```
    parser.add_option("-l", "--legacy", dest="usb", help="open a connection on the
specified USB device (0 = USB0)")
    parser.add_option("-u", "--user", dest="user", default=DEFAULT_USER,
help='specify an alternative SNAP Realm user name(default is "public")')
    parser.add_option("-p", "--password", dest="password",
default=DEFAULT_PASSWORD, help='specify an alternative SNAP Realm password (default
is "public")')
```

Here we have told the OptionParser what each of the allowable command line options is. Note that the code has not actually parsed anything yet.

```
    (options, args) = parser.parse_args()
    print options
    print args
```

The call to parser.parse_args() actually does the command line parsing, based on the previously specified rules. Note that the two print statements are just left over debugging code. It is not necessary to show the command line arguments in order to act on them.

```
    funcdir = {"echo": echo}
    comm = snap.Snap(funcs=funcdir)
```

If you have read through the earlier example in this manual, this should be pretty recognizable. We are stating that the “echo()” function is allowed to be externally callable, and we are creating a SNAP instance so that can happen.

```
    my_user = options.user
    my_password = options.password
```

Here we are just transferring the command line options into the corresponding global variables.

```
    if options.no_listener:
        print "Listening for incoming TCP/IP connections has been disallowed by
user"
    else:
        print "Listening for incoming TCP/IP connections"
        comm.accept_tcp(auth_info=my_server_auth)
```

Unless the user has told us not to (using the --deaf command line option), start listening for incoming TCP/IP connections.

```
    if len(args) != 0:
        # Establish requested outbound TCP/IP connections
        for arg in args:
            print "Establishing a TCP/IP link to "+arg
            comm.connect_tcp(arg, auth_info=my_client_auth)
```

If any IP addresses were specified on the command line, try to connect to other *SNAP Connect* instances running at those IP addresses.

```
    # Did the user ask us to open a normal serial port connection (COM1-COMn)?
    if options.comport != None:
        port = int(options.comport)
        print "Opening a serial connection on COM%d" % (port+1)
        try:
            comm.open_serial(snap.SERIAL_TYPE_RS232, port)
        except Exception, e:
            print "Invalid COM port specified"
```

```
    # Did the user ask us to open a connection to a SS200?
    if options.ss200 != None:
        port = int(options.ss200)
        print "Opening a serial connection to SNAPstick%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK200, port)
        except Exception, e:
            print "Invalid SS200 device specified"

    # Did the user ask us to open a connection to a legacy Synapse USB device?
    if options.usb != None:
        port = int(options.usb)
        print "Opening a serial connection on USB%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK100, port)
        except Exception, e:
            print "Invalid USB port specified"
```

Notice that each of the above serial connections had to specify the correct type of connection. Also notice the use of a try/except block to allow the program to continue execution, regardless of the outcome of any particular `open_serial()` call.

```
    comm.loop()
```

As pointed out in numerous places throughout this manual, most of the *SNAP Connect* functionality will not take place unless you tell the library code to actually run. Here we are handing over control of the program to the library by using the `loop()` function. If we needed to do other ongoing processing, we would have combined that processing with a call to `comm.poll()` inside of a "while 1" loop.

```
if __name__ == '__main__':
    logging.basicConfig(filename='simpleSnapRouter.log', level=logging.DEBUG,
format='%(asctime)s %(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S')
    main()
```

Nothing here that has not been shown in the other examples: a logging file is created, and then `main()` is executed.

4. Additional Examples

Throughout this manual various “tutorial” examples are presented, including detailed walk-throughs of how the source code works.

Additional examples of SNAP Connect usage are provided in the file `examples.zip`.

Time and space constraints prevent providing a detailed code walk-thru of every example application. Each example is in its own subdirectory, and each one comes with a `README.txt` file to which you should refer.

The following example programs are provided as-is. You are welcome to use them as starting points for your own custom applications, but many of them incorporate third party components or services not under Synapse control. As such, we are unable to provide individual support for them.

These examples are listed in roughly increasing order of complexity.

DisplayInfo – Sanity Check and Version Numbers Display

This example was almost considered “too simple to include”, *but* we have had repeated requests from users for exactly this sort of thing, so here it is. At a minimum, you can use this example to confirm that *SNAP Connect* is correctly installed on your system, since it requires **no** bridge hardware or serial ports of any kind.

What It Does

Example program `DisplayInfo.py` just creates a *SNAP Connect* instance and displays the Python version number, the *SNAP Connect* version number, the SNAP Address, and the current encryption setting.

How To Run This Example

In a console you can simply execute this using a python 2.6 or 2.7 interpreter:

```
python DisplayInfo.py
```

```
Python version is 2.7.9 (default, Dec  10 2014, 12:24:55) [MSC v.1500 32 bit
(Intel)]
Permanent license created on 2015-09-15 23:01:39.808000 for 000020
SNAP Connect version number 3.4.0
My SNAP Address is 00.00.20
Encryption is set to None
```

Example Files

The following files can be found in the DisplayInfo directory.

`README.txt` – describes the example in more detail

`DisplayInfo.py` – the sole source file.

EchoTest – Simple Benchmark

Refer to this example if you want to see how to maximize throughput of your SNAP network. It demonstrates use of the `HOOK_RPC_SENT` event, and shows one way to implement receive timeouts.

What It Does

EchoTest.py polls a predefined node a predefined number of times. It counts the number of responses received, times the entire test, and displays the results when finished.

```
10 queries, 10 responses in 812 milliseconds
```

How To Run This Example

```
python EchoTest.py
```

There are no command line arguments; to change the characteristics of the test (for example, which node to poll), you must edit the source code.

For an example of command line argument processing refer to `simpleSnapRouter.py`, covered earlier.

Example Files

The following files can be found in the EchoTest directory.

`README.txt` – describes the example in more detail

`EchoTest.py` – the sole source file to this example

SPY File Upload

The source code form of a SNAPpy script is a PY (“.py”) file. The compiled (binary) form of a SNAPpy script is a SPY (“.spy”) file. Just as *Portal* can, *SNAP Connect* applications can upload a replacement SPY file into an embedded SNAP Node.

What It Does

`SpyUploader.py` uploads a SPY file into a SNAP Node.

This example code serves two purposes:

1. It demonstrates a *SNAP Connect* application uploading a new .SPY file into an embedded SNAP Node.
2. It provides an easy to use API that “wraps” that same functionality, so that you can re-use the example code in your own application.

How To Run This Example

```
python SpyUploader.py
```

SpyUploader will upload (send) the pre-specified SPY file to the pre-specified SNAP Node.

There are no command line arguments; to change the characteristics of the demo (for example, what node to upload, or what SPY file to send to it), you must edit the source code.

Example Files

The following files can be found in the SpyUpload directory.

`README.txt` – describes the example in more detail

`SpyUploader.py` – the sole source file.

Note – This is both a stand-alone demo as well as an importable library that you can re-use in your own SNAP Connect applications.

`shortBlinkRF100_2.4.9.spy` – just an example SPY file to upload

As you might guess from the name, this file was compiled for a Synapse RF100 module, running firmware version 2.4.9.

If you have different hardware or a different version of SNAP firmware, just use *Portal* to create a different SPY file.

For more information on creating SPY files (and on *Portal* in general) please refer to the ***Portal User Guide***.

BulkUpload – Uploading SPY files to multiple nodes

In many cases, large numbers of nodes will all need to have their SNAPpy scripts or network configurations changed at the same time. BulkUpload can be used to simplify the process of managing nodes by performing multiple operations on all MAC addresses in a provided list.

What It Does

`bulkUpload.py` provides a command line tool for uploading SPY files, disabling multicast forwarding, changing the channel, and changing the network ID of lists of nodes.

How To Run This Example

`python bulkUpload.py` [arguments]

bulkUpload takes a series of command line arguments that determine what action will be performed on the list of nodes.

`--help, -h` : Show a help message describing all command line arguments

`--encryption, -e` : Specify the encryption type. Use 1 for AES or 2 for Basic Encryption.

`--encryptionKey, -k` : The encryption key to be used

`--macfile, -m` : The file containing the list of MACs to be operated on

`--spyfile, -s` : The SPY file to upload

`--comport, -b` : The COM port for connecting to the bridge: 'COM1', 'USB100', or 'USB200'

`--column, -c` : The column containing MAC addresses when using a CSV MAC file

`--find, -f` : Search all channels and network IDs for the given MAC addresses

`--network, -n` : Change the network ID and channel: Ex: 1,0x1234

`--nofwd, -x` : Disable multicast forwarding on the provided nodes

`--retry, -r` : The number of retries for each RPC message (Default: 3)

`--timeout, -t` : The number of seconds before a timeout occurs (Default: 3)

`--remoteConn, -z` : *SNAP Connect* IP address for forming remote connections

`--remoteBridgeAddrress, -j` : The address of the remote bridge attached to *SNAP Connect*

Example Files

The following files can be found in the BulkUpload directory.

`README.txt` – describes the example in more detail

`bulkUpload.py` – the sole source file.

`MacList.csv` – An example CSV file containing a list of MAC addresses in column 3

`MacList.txt` – An example text file containing a list of MAC addresses

TcpRawLink – Extending SNAP Data Mode over raw sockets

Embedded SNAP Nodes can exchange DATA MODE (AKA TRANSPARENT MODE) packets over radio and serial links, and with the help of one or more SNAP Connect nodes, their reach can even extend over the Internet.

But what if you have a non-SNAP TCP/IP application, such as MODBUS TCP?

What It Does

This example implements a "TCP server" so that non-SNAP devices that have TCP/IP "socket" access can send and receive SNAP TRANSPARENT DATA (DATA MODE) packets over SNAP networks.

How To Run This Example

```
python tcpRawLink.py
```

This program does not take any command line parameters. It creates a "listener" on port 3000 that you can then establish socket connections to.

Once connected, any data you send over that socket will be multicast over the SNAP network. Any DATA MODE packets received from that same SNAP network will be sent over that same socket connection to your application.

For a more detailed explanation, refer to the `README.txt` file.

Example Files

The following files can be found in the TcpRawLink directory.

`README.txt` – describes the example in more detail

`tcpRawLink.py` – the actual application source code - implements a TCP server on port 3000.

`e10Bridge.py` – This is a SNAPpy script that you can upload into your E10 Bridge node if you want LED A on the E10 to "blink" with radio traffic (optional).

You could reuse this technique with other applications; this script does not care what the source of the radio traffic is.

Extra Example Files

`S999snap` – this is meant for the /etc/init.d directory on a SNAP Connect E10. It replaces the existing file, and launches the tcpRawLink demo instead of the default E10 demo (`UserMain.py`).

`remote_daemon.py` – Linux systems can use this to run tcpRawLink.py as a background process (daemon). This file is used by `S999snap`.

FirmwareUpgrader – Creating a GUI with wxPython

SNAP Connect allows you to create applications that interact with SNAP networks, but user interaction is often limited to configuration files, command line arguments, and entering data into a terminal. Luckily, there are numerous Python libraries that can create graphical user interfaces (GUIs) to display information in a more accessible fashion and allow the user more interaction and control over the *SNAP Connect* application.

wxWidgets is a toolkit for creating GUIs that can run on multiple platforms. **wxPython** provides Python bindings that allow you to create GUI elements directly from a Python script.

What It Does

This example demonstrates how to combine the wxPython library and *SNAP Connect*. It constructs a GUI that lets you configure your bridge node, encryption type, firmware image, and target node. The GUI also provides real-time status updates to indicate how the upgrade is progressing.

This example also demonstrates how the firmware upgrade API can be used in an application.

How To Run This Example

```
python FirmwareUpgrader.py
```

This program does not take any command-line parameters. It creates a GUI with inputs to configure the bridge and encryption.

After the GUI is started, logging information is displayed on the console and output to a log file in the same directory as the application.

Additional comments are added to the source to better explain how *SNAP Connect* is tied into the **wxPython** interface.

For a more detailed explanation, refer to the `README.txt` file.

Example Files

The following files can be found in the FirmwareUpgrader directory.

`README.txt` – describes the example in more detail

`FirmwareUpgrader.py` – the actual application source code – Creates a GUI for performing over-the-air firmware upgrades.

You could reuse these techniques with other applications. The GUI elements could be altered to perform other basic operations or provide application-specific functionality.

RobotArm – Example SNAP Connect Web Server

A common customer request is “How can I create an application that is controllable through my Web Browser?”

What It Does

This example application combines *SNAP Connect* with the Tornado web server library, resulting in a program (`robotServer.py`) that interacts with web browsers on one side (via Web Sockets), and interacts with SNAP devices on the other (via *SNAP Connect* of course).

Commands made by the user in the web browser are sent from the web browser to `robotServer.py`, which converts them into SNAP RPC calls over the wireless network. Remote data reported from the SNAP nodes is received by `robotServer.py`, which updates the web page.

How To Run This Example

To run the full demo, you will need a robot arm to control. Look in file `RobotArmHardware.pdf` for details on how to modify an off-the-shelf robot arm to be controlled by a SNAP Engine. (For ease of connections, our

example makes use of a Synapse SN171 Proto Board.) You then need to load the supplied SNAPpy script "`robotArm.py`" onto that Proto Board.

If you don't have a robot arm, you can modify the script to blink LEDs or control something else of your choosing.

How To Run The Web Server Portion Of This Example

Edit file local_overrides.py and replace the line

```
serial_port = 'COM5'
```

with the correct serial_port for your computer's connection to the ProtoBoard. Then do

```
python robotServer.py
```

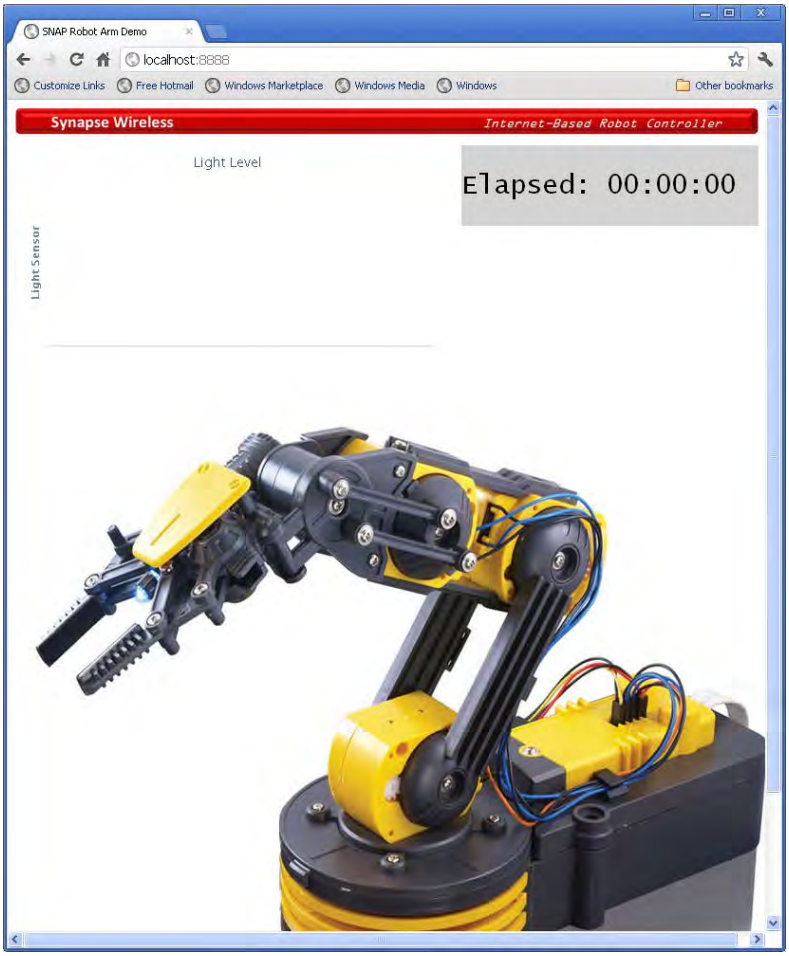
With `robotServer.py` running on your computer, you should be able to direct your web browser to http://localhost:8888 and see the "SNAP Robot Arm Demo" web page in your web browser.

NOTE – The demo requires a modern web browser that supports HTML5 Web Sockets.

If your browser is too old, you will get a message like:

```
Error
Your browser does not support HTML5 Web Sockets
```

If your browser is compatible, you should see a page like this:



If you move your mouse cursor over the image of the robot arm, you will see various arrows appear. You can click and drag these arrows to move the nearby portion of the arm.

The arm can be moved at the base (clockwise/counterclockwise), shoulder (up/down), elbow (up/down), and gripper (open/close).

You can also click to turn the light (located inside the gripper) on and off.

Example Files

The following files can be found in the RobotArm directory.

`README.txt` – describes the example in more detail

`templates\index.html` – defines the "structure" of the web page

`static\*.png` – these files define all the images of the web page

`static\main.js` – this file implements the interactivity of the web page

`static\HighCharts\*.*` - these files implement the charting capability

`static\jquery-1.6.2.min.js` – the JQuery support library

`static\jquery-ui-1.8.16.custom.min.js` – the JQuery UI library

These last three are third-party packages, and are used unmodified. For more information see `README.txt`.

The above files comprise the "web page" portion of the demo. This next file implements the actual web server and application.

`robotServer.py` – the actual web server

This program (`robotServer.py`) interacts with main.js over a Web Sockets interface. It interacts with the SNAP network (multicast RPC calls) over *SNAP Connect*.

`tornado\*.*` - this is the Tornado web server library (www.tornadoweb.org).

This third-party library is used unmodified, and is included as a convenience.

Hints and Tips

The following are some "lessons learned" in creating the example applications for this User Manual.

Make sure you know the correct SNAP Address for your SNAP Connect Application

The range of *possible* SNAP addresses for your application to use is determined by your SNAP License.

In the case of the *SNAP Connect* E10, the hardware represents your license. You can find the correct SNAP addresses by looking at the sticker on the bottom of the unit. (SNAP Connect will use your E10's Ethernet MAC address. The SNAP MAC address is for the SNAP Module in the E10, not SNAP Connect.)

When running *SNAP Connect* on a PC, the license is in the form of a license file.

- Default filename of `License.dat` (although you can override this, see `__init__()`).
- Default location is "where your application resides" (in other words, in the current working directory for your program).

A single `License.dat` file can contain multiple addresses to choose from. (This is determined when you purchase the license). The license file that comes with *SNAP Connect* contains only one license, with address 000020.

When the *SNAP Connect* library reads the license file, it will output a message similar to:

```
Permanent license created on 2011-01-25 21:59:12.640000 for 4B425A
```

The address shown (4B425A in the example) is the one you will need to use *from other nodes* when making unicast RPC calls into your running *SNAP Connect* application.

For example, if your *SNAP Connect* application provides a `logTemperature()` function, and when it launched it displayed the "Permanent license" message shown above, then other nodes would use:

```
rpc('\x4B\x42\x5A', 'logTemperature', value)
```

If by mistake you do something like…

```
rpc('some-other-address-here', 'logTemperature', value)
```

…then it will look like your *SNAP Connect* application isn't working (no temperature readings will be getting logged) when in reality your other nodes just *aren't talking to it*.

Make sure each SNAP Connect application has a unique SNAP Address

If you want to run more than one *SNAP Connect* application at the same time, you will need to license multiple SNAP addresses. Attempting to assign the same address to more than one *SNAP Connect* application at a time will result in communications failures.

Multiple applications can share the same *multi-address* `License.dat` file (or multiple copies of the same License.dat file), but be sure to specify *which* of the licensed addresses from within that license file you want each application to use. If you don't specify an address, the library will use the *first* address from within the license file. If several *SNAP Connect* instances do this, they will all be using the first address and there will be communications failures.

Make sure you know the correct serial port for your external devices

For example, it does no good to open a serial connection to COM1 when your external device is attached to COM2.

It's important to specify the correct *type* of serial port too: don't open a connection to a `SERIAL_TYPE_RS232` when you have a `SERIAL_TYPE_SNAPSTICK200` plugged in. (This mistake is less common.)

Make sure that external bridge device really is available

For example, are you already connected to that device using Portal or some other application, such as a terminal program? (Or for advanced users, do you already have *another SNAP Connect* application connected to that device?)

Don't forget to call loop() or at least poll()

The library code does not act on queued requests until you give it a chance to perform its background processing. You can call `poll()` repeatedly (intermingled with other processing), or you can just call `loop()` and turn over control to the library completely; but calling *neither* is a mistake.

Don't call poll() when you mean to call loop()

Function `poll()` only runs "one cycle" of the background code. It then returns control to the caller. If you mean for processing to continue, and you don't want to call `poll()` repeatedly (for example, from within your own "while loop"), then call `loop()` instead.

Adjust the logging levels to meet your needs

Many of the provided examples set the logging "level" (verbosity) to the maximum level. This provides the most information about what SNAP Connect is doing, but it also slows down performance. Once your application is working as you intend, you might consider changing the logging level. Refer to section String Constants used with Logging later in this manual.

Be very careful if you use Threads!

The *SNAP Connect* libraries are intended for single-thread usage only. You can use *SNAP* Connect from a multi-threaded application, but all of the accesses to the library (all API calls, etc.) must be done from a single thread.

Assuming the code "snippets" shown below are running in separate threads:

Incorrect usage

(snippet from thread 1)

```
snap_connect.rpc(nodeAddr, 'foo')
```

(snippet from thread 2)

```
snap_connect.rpc(nodeAddr, 'bar')
```

Code like the above will not work! The use of multiple threads will corrupt data internal to *SNAP Connect,* and interfere with correct serial port operation!

Correct Usage

(snippet from thread 1)

```
def send_foo():
      snap_connect.rpc(nodeAddr, 'foo')
snap_connect.scheduler.schedule(0, send_foo)
```

(snippet from thread2)

```
def send_bar():
      snap_connect.rpc(nodeAddr, 'bar')
snap_connect.scheduler.schedule(0, send_bar)
```

Code like the above will protect *SNAP Connect*'s internal data structures. Scheduling functions to run with a delay of 0 causes the function to run as quickly as if you had invoked it directly (for functions that use SNAP processing). However invoking them through the apy scheduler causes them to process from the same thread, preserving internal continuity.

Don't forget the other SNAP tools

Need to know what your *SNAP Connect* application(s) and other SNAP Nodes are saying to each other? Fire up a SNAP Sniffer!

Need a way to invoke commands manually (on demand)? Use the Portal command line! Running Portal with its own bridge node can provide a window into your complete network.

5. *SNAP Connect* API Reference – The Functions

Using *SNAP Connect* in your Python application is designed to be as easy as writing a SNAPpy script for your SNAP nodes. As you look through the API you will notice how similar the API for a SNAPpy script is compared to using *SNAP Connect*. The API was designed to follow the SNAPpy API as closely as possible, only deviating where absolutely necessary. Some API function names are formatted differently in *SNAP Connect* than in SNAPpy to follow Python's recommended style guidelines. There are, however, alias functions, shown below after a "|" character, that are formatted the same as SNAPpy's API for convenience. You can use either style in your *SNAP Connect* scripts.

__init__

```
__init__(self, license_file=None, nvparams_file=None, funcs=None, scheduler=None,
addr=None, rpc_handler=None)
```

Initialize a *SNAP Connect* instance.

`license_file` : The full path to a *SNAP Connect* license file. The default value of None indicates that the license file is named License.dat and is located in the present working directory, which will typically be the same directory that holds your Python code files.

`nvparams_file` : The full path to a *SNAP Connect* NV parameters file. The default value of None indicates that the file is named nvparams.dat and is located in the present working directory, which will typically be the same directory that holds your Python code files.

`funcs` : The callable functions for this instance to expose to the SNAP network (default None). Any given function dictionary should only be passed to a single instance of *SNAP Connect*. If you leave the parameter blank (or explicitly pass None), all functions defined in your program at the same level as your call to __init__ will be available to the RPC interface.

`scheduler` : Internally used by *SNAP Connect*

`addr` : The SNAP address to use from the license file. If nothing is specified, *SNAP Connect* uses the first address in the license file. The format of this parameter is a three-character byte string, e.g., '\x86\xa2\x5c' represents SNAP Address 86.A2.5C.

`rpc_handler` : Internally used by *SNAP Connect*

This function returns the instance of the created object.

This function might raise the following exceptions:

RuntimeError("Non-licensed address provided") – For example, the License.dat file is for address 11.22.33 but you have asked to "be" address 22.44.66.

RuntimeError("Invalid license found") – The license file specified is invalid. If you don't specify a license file name, the default of "License.dat" is assumed.

IOError("Unable to load NV params file: <filename>") – the expected NV parameters file could not be loaded. If you specify a file, it is required to be present and valid. If you do not specify a file, then the default of "nvparams.dat" is assumed, and an empty file with default values will be generated if the file is not found.

RuntimeError("Unable to determine callable functions") – you must provide a dictionary of callable functions, even if it is an empty one, or None to indicate that all functions in your application are callable.

ImportError("Unable to find PyCrypto library required for AES support") – you have enabled AES-128 encryption, but you have not provided the required PyCrypto library (refer back to the Installation section of this manual).

ValueError("Unknown encryption type specified: <encryption type>") – valid choices are NONE (0), AES128 (1), and BASIC (2).

accept_sniffer

`accept_sniffer(self)`

Start allowing remote sniffer connections over TCP.

This function returns None.

This function might raise the following exception:

RuntimeError("You must be accepting TCP connections to call accept_sniffer()") – You must make a call to `accept_tcp()` prior to accepting remote sniffer connections over TCP.

Note – Incoming remote sniffer connections will be using a different realm for the authentication function than regular TCP connections. See the section **SNAP Connect API Reference – Authentication Realms** for more details.

See also functions `accept_tcp()`, `connect_tcp()`, `stop_accepting_sniffer()`, and `stop_accepting_tcp()`.

accept_tcp

```
accept_tcp(self, auth_info=server_auth, ip='', port=48625, tcp_keepalives=False,
forward_groups=None)
```

Start listening for and accepting remote IP connections.

`auth_info` : The function to call when authenticating remote credentials (defaults to public credentials). If providing a custom function to call, it should have the signature "`server_auth(realm, username)`" where the two string arguments are supplied by the connecting instance and the function returns the appropriate password. See the SNAP Connect API Reference – Authentication Realms section for more details on the realm parameter.

`ip` : The IPv4 address to listen on for remote connections (default all addresses)

`port` : The IP port number to listen on for remote connections (default 48625)

`tcp_keepalives` : Enable or disable TCP keepalives (default False)

`forward_groups` : Multi-cast groups to forward onward through this interface; the default value of None indicates that the interface will use the value specified in NV Parameter 6

This function returns None.

The `forward_groups` parameter can be important if you have a radio network that uses multicast traffic, but you do not want those packets to propagate over the Internet.

This function might raise the following exception:

ValueError("auth_info is not callable") – Parameter auth_info can be unspecified (in which case the default server_auth() routine is used), but if you provide a value for this parameter it must be a function that *SNAP Connect* can invoke.

Establishment of an actual connection can trigger a HOOK_SNAPCOM_OPENED event.

If the connection later goes down it can trigger a HOOK_SNAPCOM_CLOSED event.

See also functions `connect_tcp()`, `disconnect_tcp()`, and `stop_accepting_tcp()`.

add_rpc_func

```
add_rpc_func(self, rpc_func_name, rpc_func, allow_alternate_key = False)
```

Adds a function to the existing "RPC dictionary".

`rpc_func_name` : This is the name the new function will be callable by via RPC. It does not have to match the actual function's name.

`rpc_func` : This must be a Python callable, and represents the actual function to be invoked.

`allow_alternate_key` : Defaults to False if unspecified, but if True adds the specified *rpc_func_name* to a "whitelist" of functions that can also be remotely invoked with alternate encryption (an alternate encryption key). See also function `rpc_alternate_key_used()`, which is how you can detect that this has actually taken place.

When the *SNAP Connect* instance is first instantiated by your application code, you pass it a Python dictionary containing all of the function names that you want to be able to invoke from other SNAP nodes. This function lets you add *additional* functions to that dictionary after-the-fact.

This function returns True if the function was successfully added to the RPC dictionary. It returns False if the function could not be added because one with the same name already exists in the dictionary. (You can use this function to **add** a new function but you cannot use it to **replace** an existing function.)

allow_serial_sharing

```
@staticmethod Snap.allow_serial_sharing(allow_sharing=True)
```

Enable or disable sharing of serial connections between *SNAP Connect* instances.

Note –*SNAP Connect* instances must be running in the same Python interpreter session in order to share serial connections. *SNAP Connect* instances running in different Python interpreter sessions will not be able to share serial connections.

`allow_sharing`: True if serial connection sharing should be enabled or False if it should be disabled. This parameter is shared across all *SNAP Connect* instances. This defaults to **True**.

If serial connection sharing is enabled, all open serial connections will be *shared* when a new *SNAP Connect* instance is constructed. Serial connections opened after a *SNAP Connect* instance is constructed will *not* be shared.

Shared serial connections will be able to receive broadcast messages and route for all connected *SNAP Connect* instances.

This function returns None.

Note – This is a static function on the *Snap* class. This should not be invoked from a created *SNAP Connect* instance and should be called like `snap.Snap.allow_serial_sharing(True)`

See also function `open_serial()`.

cancel_upgrade

```
cancel_upgrade(self, upgrade_addr)
```

Cancel an over-the-air firmware upgrade.

`upgrade_addr` : The SNAP network address of the remote node to cancel the firmware upgrade

This function returns **True** if the upgrade was canceled. This function returns **False** if no upgrade is currently running for the given address or the upgrade has already completed.

This function can result in a `HOOK_OTA_UPGRADE_COMPLETE` event if an upgrade was successfully canceled.

See also function `upgrade firmware()`.

close_all_serial

```
@staticmethod Snap.close_all_serial()
```

Close all opened serial connections for all *SNAP Connect* instances.

This is equivalent to calling the `close_serial` function for every serial connection created with `open_serial`.

This function returns None.

Note – This is a static function on the *Snap* class. If multiple instances of *SNAP Connect* are running in the same application, they will all have their serial connections closed.

See also functions `open serial()` and `close serial()`.

close_serial

```
close_serial(self, serial_type, port)
```

Close the specified serial port if open.

serial_type : The type of serial interface to close (RS-232=1, Legacy USB=2, Synapse SS200=5)

port : The port **of that particular type** to close, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example "/dev/ttys1".

This function returns None.

This function can trigger a HOOK_SERIAL_CLOSE event.

See also function `open serial()`.

connect_tcp

```
connect_tcp(self, host, auth_info=client_auth, port=None, retry_timeout=60,
secure=False, forward_groups=None, tcp_keepalives=False, cache_dns_lookup=False,
sniffer_callback=None)
```

Connect to another SNAP node over TCP/IP.

`host` : The IPv4 address or hostname of the other SNAP node as a dotted string (e.g., "192.168.1.1")

`auth_info` : The function to call when authenticating remote credentials (defaults to public credentials). If providing a custom function to call, it should have the signature "`server_auth(realm, username)`" where the two string arguments are supplied by the connecting instance and the function returns the

appropriate password. See the SNAP Connect API Reference – Authentication Realms section for more details on the realm parameter.

`port` : The IP port number to connect to (defaults to port 48625 if None given)

`retry_timeout` : A timeout in seconds to wait before retrying to connect (default 60)

`secure` : A boolean value specifying whether SSL encryption is enabled for this connection (default False)

> **Note:** *SNAP Connect* can initiate an SSL connection but cannot accept an SSL connection. Some type of proxy is necessary to decrypt and reroute received traffic in order to form an encrypted connection between *SNAP Connect* instances. For example, **stunnel** (www.stunnel.org) could be used to receive encrypted data on a port, decrypt the data, and forward it to the port *SNAP Connect* is using to `accept_tcp`. When secure is set to True, the port will default to **443**.

`forward_groups` : Multi-cast groups to forward onward through this interface; defaults to using the value specified in NV Parameter 6.

`tcp_keepalives` : Enable TCP layer keepalives (default False.)

> **Note:** When using `tcp_keepalives` you may experience conflicts with certain firewall configurations, which may close the connection.

`cache_dns_lookup` : Only perform a DNS lookup once for the host being connected to (default False). This option was added because some DNS servers are extremely slow.

`sniffer_callback`: Callback used for making remote sniffer connections to *SNAP Connect* instances that have called both `accept tcp()` and `accept sniffer()`. When this callback is used, regular traffic will not pass across this TCP connection, and the accepting end of the connection will forward all received and transmitted packets to this interface. This function should have the signature "`sniffer(descriptor)`". See the section SNAP Connect API Reference – Sniffer Descriptors for more details on the descriptor objects that are passed to local and remote sniffers. The sniffer is NOT sniffing over-the-air packets. This callback is only passed packets which are received and transmitted over *SNAP Connect* serial and TCP connections.

SNAP Connect only supports connecting to IPv4 addresses. If a hostname is provided it must be able to be resolved to an IPv4 address.

The forward_groups parameter can be important if you have a radio network that uses multicast traffic, but you do not want those packets to propagate over the Internet.

> **NOTE** – Enabling tcp_keepalives can increase the amount of TCP/IP traffic your SNAP network generates. This might be an issue if you are using a cellular modem for your network connectivity.

This function returns None.

Establishment of the actual connection can trigger a `HOOK_SNAPCOM_OPENED` event.

If the connection later goes down it can trigger a HOOK_SNAPCOM_CLOSED event.

See also functions `accept tcp()`, `disconnect tcp()`, `accept sniffer()`, and `stop accepting tcp()`.

data_mode

`data_mode(self, dst_addr, data)`

Sends a transparent (aka data) mode packet to the specified SNAP network address.

dst_addr : The SNAP network address of the remote node, as a three-character string

data : The data to send

This function returns an identifier for the packet sent, equivalent to the value returned from a call to get_info(9). This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can result in a HOOK_STDIN event at the node specified by dst_addr.

See also mcast data mode().

directed_mcast_rpc

directed_mcast_rpc(self, conn, port, group, ttl, func_name, *args)

Makes a Remote Procedure Call, or RPC, using multicast messaging. This means the message could be acted upon by multiple nodes. The multicast message will only be directed out one interface instead of broadcast across all open TCP and serial connections.

Note that despite the similarity of the names, the result of this function is very different from that of the dmcast rpc function. This function restricts which interface will be used to send a standard multicast message, while the dmcast rpc function sends a message across all interfaces (subject to the groups parameter forwarding compatibility).

conn : For serial connections, this is the serial type passed to open serial(). For TCP interfaces, this value should be a string for the IP address of the interface.

port : The TCP or serial port number of the interface that should transmit the message.

group : Specifies which nodes should respond to the request, based on each node's Multicast Processing Group NV parameter.

ttl : Specifies the Time To Live (TTL) for the request.

func_name : The function name to be invoked.

args : Any arguments for the function specified by func_name. See mcast rpc() for more details.

This function returns an identifier for the packet sent, equivalent to the value returned from a call to get info(9). This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can trigger a HOOK_RPC_SENT event.

See also functions mcast rpc() and dmcast rpc().

disconnect_tcp

disconnect_tcp(self, host, port=48625, all=False, retry=False)

Disconnect from the specified instance.

host : The IP address or hostname of the other instance

port : The IP port number (default 48625)

all : Disconnect all connections matching the criteria (default False)

retry : Disconnect, but then retry connecting to the same host and port (default False)

This function returns True if the specified connection was found and closed, otherwise False.

This function can result in one or more HOOK_SNAP_CLOSED events being generated.

See also functions `accept_tcp()`, `connect_tcp()`, and `stop_accepting_tcp()`.

dmcast_rpc | dmcastRpc

`dmcast_rpc(self, dst_addrs, group, ttl, delay_factor, func_name, *args)`

Makes a Directed Remote Procedure Call, or RPC, using multicast messaging. This means the message could be acted upon by multiple nodes. Unlike a standard multicast, however, the message will only be acted upon by nodes explicitly listed in `dst_addrs`. Unlike an addressed RPC call, this directed multicast does not make use of routing or packet acknowledgement. Other nodes in the mesh network will forward the message (subject to their Multicast Forwarded Groups settings in NV Parameter 6) when there is sufficient TTL to do so. There is no route discovery performed, and there are no retries.

Note that though they have similar names, `dmcast_rpc` functions very differently from `directed_mcast_rpc`. The `directed_mcast_rpc` function allows you to restrict the interface used to transmit your message, but the message sent is a “standard” multicast. This function forwards through all interfaces that would normally forward the message (subject to the multicast group forwarding settings specified when the interface connection is opened), but will only be acted on by the node(s) specified, and then only if the multicast group specified for the call matches the node’s Multicast Processed Groups setting in NV Parameter 5.

`dst_addrs` : The `dst_addrs` parameter should be a string containing concatenated addresses for any nodes you wish to act on the directed multicast. For example if you have nodes with addresses 01.02.03, 04.05.06, 07.08.09, and 0A.0B.0C, the parameter should contain "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c ". If you provide a string that is not a multiple of three characters in length, the `dmcast_rpc` call will fail and no message will be sent to any node.

If you provide an empty string ("") for this parameter, all nodes that receive the message that would otherwise act on the message (subject to the `groups` parameter and the existence of the function in the node’s script) will act on the request as though the call were a regular `mcast_rpc()` call. However in this case, added features available only for directed multicast (such as information available through several `get_info()` calls) are also available.

`group` : Specifies which nodes should respond to the request, based on each node’s Multicast Processing Group NV parameter.

`ttl` : Specifies the Time To Live (TTL) for the request.

`delay_factor` : Parameter `delay_factor` provides a mechanism to allow receiving nodes to stagger their responses to the request. The parameter should be a one-byte integer specifying the amount of time, in milliseconds, that should pass between node responses among the nodes targeted by the request. For example, if the `dst_addrs` parameter contains "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c " and the `delay_factor` parameter contains 40, any radio traffic generated in response by node 01.02.03 would be released immediately, radio traffic generated by 04.05.06 would be queued and held for 40 ms, radio traffic generated by 07.08.09 would be delayed for 80 ms, and radio traffic generated by 0A.0B.0C would be held for 120 ms before release.

Note that the function on the target node is executed without delay. The delay only applies to radio communications directly invoked by the called function. No delay is applied to serial communications from the receiving node. Thus, an instance of SNAP Connect, which has only serial interfaces, is not affected by this parameter.

Setting this parameter to zero allows all receiving nodes with radio interfaces to respond immediately, which may cause packet loss due to interference.

A check of `get_info(25)` in the called function on the receiving node returns the `delay_factor` value specified, but also tells the receiving node to ignore the transmission delay that would normally be enforced.

`func_name` : The function name to be invoked.

`args` : Any arguments for the function specified by `func_name`. See `mcast_rpc()` for more details.

This function returns an identifier for the packet sent, equivalent to the value returned from a call to `get_info(9)`. This identifier can later be used in conjunction with the `HOOK_RPC_SENT` handler.

This function can trigger a `HOOK_RPC_SENT` event.

See also functions `mcast_rpc()`, `directed_mcast_rpc()`, and `rpc()`.

get_info | getInfo

```
get_info(self, which_info)
```

Get the specified system information.

`which_info` : Specifies the type of information to be retrieved (0-15 but with some gaps – not all of the "info" types from the original [embedded] SNAP Nodes apply to a PC-based application like *SNAP Connect*). For more information on the `get_info()` function, refer to the **SNAP Reference Manual**. This function returns the requested information, or None if the `which_info` parameter is invalid.

The possible values for `which_info`, and their meanings/return values are:

which_info	Meaning	Return Value
0	Vendor	Always returns 0 indicating "Synapse"
1	Type of radio	Always returns 1 indicating "no radio"
2	Type of CPU	Always returns 8 indicating "unknown"
3	Hardware platform	Always returns 7 indicating "*SNAP Connect*"
4	Build Type (debug or release)	Returns 0 if running under a debugger, Returns 1 if running standalone
5	Software MAJOR version	Example: if version is 3.2.1 get_info(5) returns 3
6	Software MINOR version	Example: if version is 3.2.1 get_info(6) returns 2
7	Software BUILD version	Example: if version is 3.2.1 get_info(7) returns 1
8	Encryption capability	Returns 1 if AES-128 is available (not necessarily enabled, just available for use) Otherwise returns 2, indicating that only "SNAP Basic" encryption is available

9	SNAP Sequence Number of most recently enqueued packet	For SNAP Connect applications you should use the value returned by the `rpc()`, `mcast_rpc()`, and `dmcast_rpc()` functions. This enumeration is only implemented to loosely match the embedded nodes
10	Multicast flag	Returns 1 if the RPC currently being processed came in via multicast; returns 0 if the packet came in via unicast
11	TTL Remaining	Returns the TTL (the "hops remaining") field of the packet currently being processed. For this to be of any use, you would have to know how many hops were originally specified
12	Tiny strings remaining	N/A to *SNAP Connect*, always returns None
13	Small strings remaining	N/A to *SNAP Connect*, always returns None
14	Size (capacity) of Route Table	N/A to *SNAP Connect*, always returns None
15	Routes stored in Route Table	Returns the number of active routes
16	Bank Free Space	N/A to *SNAP Connect*, always returns None
17	Reserved	N/A to *SNAP Connect*, always returns None
18	STDIN Hook Trigger	N/A to *SNAP Connect*, always returns None
19	Remaining Tiny Strings	N/A to *SNAP Connect*, always returns None
20	Remaining Large Strings	N/A to *SNAP Connect*, always returns None
21	Detect First Execution of Script	N/A to *SNAP Connect*, always returns None
22	Base Address of Script	N/A to *SNAP Connect*, always returns None
23	Base Bank of Script	N/A to *SNAP Connect*, always returns None
24	Is Directed Multicast	Returns a 1 if the function running was invoked by a directed multicast using `dmcast rpc()`. Returns a 0 if the function running was invoked by a hook or scheduled event, by an addressed RPC, or by a "normal" multicast (i.e., `directed mcast rpc()` or `mcast rpc()`).
25	Read Delay Factor	Returns the delay factor specified for a message sent using `dmcast rpc()`. This delay factor has no effect on a SNAP Connect-based node, because the SNAP Connect instance has only serial interfaces.

26	Address Index	Directed multicasts messages sent using `dmcast rpc()` can target multiple nodes by concatenating multiple SNAP addresses in the dst_addrs parameter. This option indicates where the address of the contextual node appears in that list, as a zero-based index.
27	Multicast Groups	Returns an integer indicating the multicast group mask specified for the packet when it was originally sent, if it was sent as a directed multicast using `dmcast rpc()`.
28	Original TTL	Returns the TTL (the "hops") field specified for the packet when it was originally sent, if it was sent as a directed multicast using `dmcast rpc()`. Combined with get_info(11), you can determine how many hops it took for the message to reach your node.

Note: The string "`snap.VERSION`" is equivalent to retrieving the major, minor and build revisions (getinfo 5, 6, and 7 respectively). For example:

```
>>> from snapconnect import snap
>>> snap.VERSION
'3.4.0'
```

load_nv_param | loadNvParam

`load_nv_param(self, nv_param_id)`

Return the indexed parameter from storage.

`nv_param_id` : Specifies which "key" to retrieve from storage. Some NV parameters may have no effect on the *SNAP Connect* instance.

For more details about the individual NV Parameters, refer to section SNAP Connect API Reference – NV Parameters later in this document.

This function returns the requested NV Parameter. Note that on some platforms the MAC Address parameter is set by the hardware (for example, on a *SNAP Connect E10*). In such cases, a request for that parameter will return the "hardware" value rather than any artificially stored value.

See also function `save nv param()`.

local_addr | localAddr

`local_addr(self)`

This function returns the three-byte binary string representing the address of the *SNAP Connect* instance. For example, a *SNAP Connect* instance running at the default address of 00.00.20 would return a Python string containing "\x00\x00\x20".

loop

`loop(self)`

This function does not return. You should only call it if your *SNAP Connect* application is "purely reactive" – for example, an application that only responds to RPC calls from other nodes.

If your application needs to take action on its own behalf (for example, if it needs to be polling other nodes but needs to perform its own independent processing between received messages), then you probably should be using the `poll()` function instead.

Calling the loop function is essentially equivalent to using this in your code:

```
from time import sleep
while True:
    poll()
    sleep(0.001)
```

The 1 ms sleep between polls prevents SNAP Connect from completely monopolizing your CPU. You can adjust this sleep duration by setting the `snap.SLEEP_TIME` global in your code to a value, in seconds. Longer sleep periods between checks consume less of your CPU, but also reduce the responsiveness of your network as it takes longer to get packets between SNAP Connect and your bridge node (or other interface).

```
snap.SLEEP_TIME = 0.1
loop()
```

See also function `poll()` and function `poll internals()`.

mcast_data_mode

`mcast_data_mode(self, group, ttl, data)`

Sends a multicast transparent (aka data) mode packet.

`group` : specifies which nodes should respond to the request, based on the receiving node's Multicast Processed Groups NV parameter

`ttl` : specifies the Time To Live (TTL) for the request

`data` : The string data to send

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the `HOOK_RPC_SENT` handler.

This function can result in a `HOOK_STDIN` event in one or more other nodes.

See also function `data mode()`.

mcast_rpc | mcastRpc

`mcast_rpc(self, group, ttl, func_name, *args)`

Makes a Remote Procedure Call, or RPC, using multicast messaging. This means the message could be acted upon by multiple nodes.

`group` : specifies which nodes should respond to the request, based on the receiving node's Multicast Processed Groups NV parameter

`ttl` : specifies the Time To Live (TTL) for the request

`func_name` : The function name to be invoked

`args` : Any arguments for the function specified by `func_name`. Note that these should be given individually (separated by commas), not bundled into a tuple. For example, if `foo()` is a function taking two parameters, use something like

```
mcast_rpc(1, 2, 'foo', 1, 2)  # <- correct!
```

instead of using something like

```
mcast_rpc(1, 2, 'foo', (1,2))  # <- wrong!
```

This is different from the way multiple parameters were handled in the original XML-RPC based *SNAP Connect*.

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the `HOOK_RPC_SENT` handler.

This function can trigger a `HOOK_RPC_SENT` event.

See also function `rpc()`.

open_serial

```
open_serial(self, serial_type, port, reconnect=False, dll_path=MODULE_PATH,
forward_groups=None, baudrate=38400)
```

Open the specified serial port for sending and receiving SNAP packets.

`serial_type` : The type of serial interface to open. Available options are:

- `SERIAL_TYPE_SNAPSTICK100`: The original USB SNAP Stick SN132 SNAP Engine carrier, or the USB interface on an SN163 demonstration board
- `SERIAL_TYPE_SNAPSTICK200`: The SNAP Stick 200
- `SERIAL_TYPE_RS232`: An RS-232 COM port

Note – SNAPSTICK100 refers to the SN132 paddle board or SN163 demonstration board, regardless of the SNAP Engine type installed. This could be an RF100, an RF200, an RF300, or any other Synapse product using the 24-pin RF Engine footprint.

`port` : The port **of that particular type** to open, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example `"/dev/ttys1"`.

`reconnect` : Close the connection and re-open (default False)

`dll_path` : Path to where the serial port driver exists on your system

`forward_groups` : Multi-cast groups to forward onward through this interface; defaults to using the value specified in NV Parameter 6

`baudrate`: The baud rate used to communicate using the serial port

This function normally returns False. If you specified `reconnect=True` and the reconnect was successful, then True will be returned.

This function will trigger a `HOOK_SERIAL_OPEN` event.

This function can raise the following exceptions:

ValueError("Serial interface type must be an integer")

ValueError("Unsupported serial interface type <type>")

Note – Linux on the E10 allows multiple processes to open the same serial port, but a SNAPConnect application must have exclusive access to the serial port. (Multiple SNAP Connect instances **can** share a serial connection, if they are created within a single application.) Ensure that only one SNAPConnect application is running at any one time. (The E10 starts a default application from `/etc/init.d/S999snap.`)

poll

`poll(self)`

Performs one cycle of background operations, including network (asyncore) and scheduling processing, then returns. By calling this function repeatedly, you can keep *SNAP Connect* communications going while still performing other processing (for example, maintaining a GUI).

If your *SNAP Connect* application has processing of its own to do (it is active rather than reactive), then function `poll()` or function `poll_internals()` is the function to use.

If your application will only be responding to incoming RPC calls from other nodes, then you might consider using the convenience function `loop()` (which just calls `poll()` repeatedly).

NOTE – In the absence of any other processing in your loop, this function provides no break from the checking and processing of SNAP events. Simply having a `poll()` call in a `while True:` loop will result in nearly complete consumption of your CPU's available processing cycles. It is recommended that you include a `time.sleep(duration)` call within your loop to reduce your CPU consumption, with a duration on the order of 0.001 seconds.

This function returns None.

NOTE – This function performs not only the *SNAP Connect*-specific processing, but also invokes `asyncore.poll()` and the `scheduler.poll()` function. For many stand-alone *SNAP Connect* applications, this is optimal.

However, some applications (such as a web server that has SNAP Connect embedded inside it) may need to maintain control of asyncore and scheduling. In those situations, function `poll()` will be trying to "do too much", and you should use function `poll_internals()` instead.

See also function `loop()` and `poll_internals()`.

poll_internals

`poll_internals(self)`

Performs one cycle of *SNAP Connect*-specific background operations processing only (no asyncore or scheduling), then returns. By calling this function repeatedly, you can keep *SNAP Connect* communications going while still performing other processing (for example, maintaining a GUI).

If your *SNAP Connect* application has processing of its own to do (it is active rather than reactive), then `poll()` is the function to use. If that additional processing involves asyncore and scheduling, you may need to use function `poll_internals()` instead of function `poll()`.

For an example of a web server application that demonstrates the use of `poll_internals()` instead of function `poll()`, refer to the "Robot Arm" example application.

This function returns None.

replace_rpc_func

`replace_rpc_func(self, rpc_func_name, rpc_func)`

Replaces an existing function in the RPC dictionary, which is the function dictionary specified when you invoke a *SNAP Connect* instance.

`rpc_func_name` : This is the name the RPC callable function to be replaced. It does not have to match the actual function's name.

`rpc_func` : This must be a Python callable, and represents the actual function to be invoked.

When the *SNAP Connect* instance is first instantiated by your application code, you pass it a Python dictionary containing all of the function names that you want to be able to invoke from other SNAP nodes. Additional RPC functions can be added by calling `add_rpc_func()`.

This function lets you *replace* functions that have already been added to the RPC dictionary (or that were already included in the dictionary when the *SNAP Connect* instance was created).

This function returns the function that was replaced if the function name already existed in the RPC dictionary. It returns None if the function could not be replaced because one with the same name did not already exists in the dictionary. (You can use this function to **replace** a new function but you cannot use it to **add** a new function.)

See also `add_rpc_func()`.

rpc

`rpc(self, dst_addr, func_name, *args)`

Sends a unicast RPC command.

Call a remote procedure (make a Remote Procedure Call, or RPC), using unicast (directly addressed) messaging. A packet will be sent to the node specified by the `dst_addr` parameter, asking that remote node to execute the function specified by the `func_name` parameter. The specified function will be invoked with the parameters specified by `args`, if any `args` are present.

`dst_addr` : The SNAP network address of the remote node, a three-character string

`func_name` : The function name to be invoked on the remote node

`args` : Any arguments for the function specified by func_name. Note that this should be given individually (separated by commas), not bundled into a tuple. For example, if `foo()` is a function taking two parameters, use something like

```
rpc('\x12\x34\x56', 'foo', 1, 2)  # <- correct!
```

instead of using something like

```
rpc('\x12\x34\x56', 'foo', (1,2))  # <- incorrect!
```

This is different from the way multiple parameters were handled in the original XML-RPC based *SNAP Connect*.

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can trigger a HOOK_RPC_SENT event.

See also `mcast_rpc()`.

rpc_alternate_key_used

`rpc_alternate_key_used(self)`

This function returns False except in the special case where the current packet was received with alternate encryption, in which case it returns True.

Note that several steps have to be taken to even allow for this possibility:

1) There has to **be** an alternate encryption key – see NV parameter 55, Alternate Encryption Key
2) Each individual function has to be explicitly "whitelisted" to be eligible for "reception via alternate encryption key" – see `add rpc func()`

See also functions `rpc source addr()`, `rpc source interface()`, and `rpc use alternate key()`.

rpc_source_addr | rpcSourceAddr

`rpc_source_addr(self)`

Originating address of the current RPC context (or None if called outside RPC).

This function returns the SNAP network address of the remote node that initiated the RPC call.

If no RPC call is currently in progress, then this function returns None.

See also function `rpc source interface()`.

rpc_source_interface

`rpc_source_interface(self)`

Originating interface of the current RPC context (or None if called outside RPC).

This function returns an Interface object representing the interface type on which the incoming RPC call was received. The Interface object can be queried for its type, as necessary.

That type is contained in the `intf_type` field of the Interface object, and will be one of the following values:

```
INTF_TYPE_UNKNOWN = 0
INTF_TYPE_802154 = 1
INTF_TYPE_SERIAL = 2
INTF_TYPE_SILABS_USB = 3
INTF_TYPE_ETH = 4
INTF_TYPE_PROXY = 5
INTF_TYPE_SNAPSTICK = 6
```

If no RPC call is currently in progress, then this function returns None.

See also function `rpc source addr()`.

rpc_use_alternate_key

`rpc_use_alternate_key(self, enable_flag)`

Control the use of the alternate encryption key when enqueueing `mcast rpc()` packets for transmission. Parameter *enable_flag* can be:

True – all subsequent `mcast rpc()` packets will be encrypted with the alternate encryption key instead of the normal key. This setting remains in effect until you call `rpc_use_alternate_key(False)`

False – the alternate key will no longer be used (this is the default setting).

The alternate RPC key does not affect non-multicast RPC traffic.

See also NV parameter 55, Alternate Encryption Key.

save_nv_param | saveNvParam

```
save_nv_param(self, nv_param_id, obj, bitmask=None)
```

Store individual objects for later access by id. Values persist through stopping and restarting SNAP Connect.

`nv_param_id` : Specifies which "key" to store the obj parameter under. Integers 0-127 are pre-assigned system IDs. Integers 128-254 are available for user-defined purposes on all SNAP nodes, but *SNAP Connect* instances allow you to use any value as a key.

`obj` : The object you would like to store for later use. Other SNAP nodes are capable of storing strings, integers, and Boolean values; but *SNAP Connect* instances can store any object that Python is capable of pickling. Refer to the Python documentation for details about the pickle functions.

`bitmask` : An integer specifying which bits of the existing integer value in an NV parameter should be updated when saving an integer value back (as the `obj` parameter) into an NV parameter. For example, if NV parameter 170 contained 0xf001, a call of save_nv_param(170, 0x05f0, 0xeac5) would leave the value 0x10c0 in the parameter.

```
Start value: 0xf001 = 1111 0000 0000 0001
New value:   0x05f0 = 0000 0101 1111 0000
Bitmask:     0xeac5 = 1110 1010 1100 0101
Final value: 0x10c0 = 0001 0000 1100 0000
```

In the above example, red digits show where the bit changed because the bitmask was set and the new value was different from the start value; *green digits in italics show where the bit did not change because even though the bitmask bit was set, the start value and the new value already had the same bit;* blue digits show where the bit did not change because the bitmask was not set, even though the new value was different from the start value; and black digits show where the bit did not change because the start and new values were already the same, regardless of the bitmask being unset.

The feature was implemented to facilitate setting of feature bit fields on SNAP Engines, which are limited to 16-bit integers. The result of using this on anything other than 16-bit integers in SNAP Connect is undefined.

If the bitmask parameter is omitted or None, the previous value of the NV parameter will be replaced with the new `obj value`.

The `save_nv_param` function returns True if the save operation was successful; otherwise it returns False.

SNAP Engine-based nodes must reboot before changes to NV parameters take effect. However parameter changes affecting the configuration of SNAP Connect instances take effect immediately. For example, you do not need to restart SNAP Connect after enabling or disabling encryption for the change to be effective.

See also function `load_nv_param()`.

set_hook | setHook

```
set_hook(self, hook, callback=None)
```

Set the specified SNAP hook to call the provided function.

`hook` : The SNAP event hook identifier. Available identifiers are:

HOOK NAME	EVENT

hooks.HOOK_SNAPCOM_OPENED	Network communications established
hooks.HOOK_SNAPCOM_CLOSED	Network communications shut down
hooks.HOOK_SERIAL_CLOSE	Serial port closed
hooks.HOOK_SERIAL_OPEN	Serial port opened
hooks.HOOK_RPC_SENT	RPC sent (or unicast retries exhausted)
hooks.HOOK_STDIN	Data received from another SNAP node
hooks.HOOK_TRACEROUTE	Trace Route results have been received
hooks.HOOK_OTA_UPGRADE_COMPLETE	Over the air upgrade has been completed
hooks.HOOK_OTA_UPGRADE_STATUS	Over the air upgrade status update
hooks.HOOK_TOPOLOGY_POLL_RESULTS	Local topology results available

`callback` : The function to invoke when the specified event occurs

The required signature for a given callback handler depends on the particular hook. Refer to section SNAP Connect API Reference – HOOKS towards the end of this document.

This function returns None.

This function can raise the following exceptions:

TypeError("Unknown hook type") – you can only specify hook values from the above list.

TypeError("Invalid callback") – the object you specify to be the handler for the specified hook must be a callable function.

start_sniffer

```
start_sniffer(self, callback)
```

This function starts a local sniffer for the calling *SNAP Connect* instance.

callback : The function invoked when a packet is transmitted or received by the local *SNAP Connect* instance. This function should have the signature "`sniffer(descriptor)`". See the section SNAP Connect API Reference – Sniffer Descriptors for more details on the descriptor objects that are passed to local and remote sniffers.

NOTE – The sniffer is NOT sniffing over the air packets. This callback is only passed packets that are received and transmitted over *SNAP Connect* serial and TCP connections.

This function returns True if the local sniffer was started or False if a local sniffer is already running.

See also function `stop_sniffer()`.

stop_accepting_sniffer

```
stop_accepting_sniffer(self, close_existing=False)
```

This function cancels any previous `accept_sniffer()` calls.

`close_existing` : In addition to not accepting any new incoming remote sniffer connections, this parameter tells *SNAP Connect* to automatically shut down any remote sniffer connections that have already been established (default False).

This function returns None.

Note – For stopping *local* sniffers created with `start_sniffer()`, use the `stop_sniffer()` function. This function is only used for remote packet sniffers.

See also functions `accept_sniffer()`, `accept_tcp()`, and `connect_tcp()`.

stop_accepting_tcp

```
stop_accepting_tcp(self, close_existing=False)
```

This function cancels any previous `accept_tcp()` calls.

`close_existing` : In addition to not accepting any new incoming connections, this parameter tells SNAP Connect to automatically shut down any connections that have already been established (default False).

If `close_existing = True`, this function can result in one or more HOOK_SNAP_CLOSED events being generated.

This function returns None.

See also functions `accept_tcp()`, `connect_tcp()`, and `disconnect_tcp()`.

stop_sniffer

```
stop_sniffer(self)
```

This function stops the local sniffer for the calling *SNAP Connect* instance.

This function returns True if the local sniffer was stopped or False if no local sniffer was already running.

Note – For stopping *remote* sniffer connections, you will need to use `stop_accepting_sniffer()`. This function is only used for local packet sniffers.

See also function `start_sniffer()`.

traceroute

```
traceroute(self, dst_addr)
```

Sends a traceroute request.

`dst_addr` : The SNAP network address of the remote node to perform a Trace Route to.

A Trace Route determines/reports a current route to the node that has the network address specified as specified `dst_addr`. This does not mean that the node cannot also be reached through other paths.

This function returns None. To actually get the "trace route report" you will need to use `set_hook(snap.hooks.HOOK_TRACEROUTE, …)` to specify a function that will receive and process the trace route list.

This function can result in a HOOK_TRACEROUTE event.

Refer to section HOOK_TRACEROUTE – Trace Route data received for more details.

upgrade_firmware

`upgrade_firmware(self, upgrade_addr, firmware_file)`

Perform an over-the-air firmware upgrade.

`upgrade_addr` : The SNAP network address of the remote node to receive the firmware upgrade

`firmware_file` : The path to the firmware file used for the upgrade

This function returns **True** if the upgrade was successfully started. This function returns **False** if an upgrade is already running for this address or there is a problem with the firmware file.

This function will result in a `HOOK_OTA_UPGRADE_COMPLETE` event.

This function can result in many `HOOK_OTA_UPGRADE_STATUS` events.

Refer to sections HOOK_OTA_UPGRADE_COMPLETE – Over the air upgrade complete and HOOK_OTA_UPGRADE_STATUS – Over the air upgrade status report for more details

NOTE – Not all nodes support over-the-air firmware upgrades. Please make sure to provide the correct type of upgrade file for the node at the address you pass to this function.

See also function `cancel_upgrade()`.

vmStat

`vmStat(self, status, arg=None, delay=0)`

This function exists in SNAP Connect to provide compatibility with some network topology functions on other SNAP nodes.

`status`: Most of the `vmStat` status codes discussed in the SNAP Reference Manual do not apply to SNAP Connect instances (such as network ID and channel, link quality, etc.). However invoking `vmStat(12)` on a SNAP Connect instance will cause the SNAP Connect instance to query its network neighbors and reply to the calling node with at least one `tellVmStat` call, each with two parameters: the first parameter will be an integer, the low byte of which will be 12; the second will be a string which will have a length a multiple of five characters.

Each set of five characters in the second parameter will be the address of a node with which the SNAP Connect instance can directly communicate in the first three bytes, and radio link quality values that have no meaning in the context of serial communications in the fourth and fifth byte.

Thus if a node invokes `vmStat(12)` on a SNAP Connect instance, it will receive one (or more, if the SNAP Connect instance has many connections open) `tellVmStat` callback identifying the nodes to which the SNAP Connect instance has a direct serial or TCP/IP connection.

`arg`: Some `vmStat` commands take an optional argument. For instance, invoking `vmstat(12, 3)` on a SNAP Connect instance will cause the SNAP Connect instance to query its network neighbors with a 3 second response window. Other `vmStat` commands interpret this argument differently, see the **SNAP Reference Manual** for details.

`delay`: The `delay` parameter is optional, and if present instructs the SNAP Connect instance to wait a random duration up to the length of the specified `delay` value (in seconds) to reply with its `tellVmStat` call. If the node calling `vmStat` on the SNAP Connect instance does so with a multicast to all adjacent nodes, this random response window reduces the likelihood that responding nodes will "step on" each other as they reply.

NOTE – If you invoke `vmStat(12)` from your SNAP Connect instance on another node, you must implement the appropriate `tellVmStat(header, data)` function in your SNAP Connect application to catch and process the data returned by that node.

NOTE – If you want to get local topology results, you can now call a function of your choosing that will be triggered by the HOOK_TOPOLOGY_POLL_RESULTS hook. See section 9 for more information.

6. *SNAP Connect* API Reference – Constants and Enumerations

This section lists and describes the numerous constants defined by the *SNAP Connect* libraries for readability.

NOTE: Most of these constants are defined by the snap module, and so need a "snap." prefix on them. For readability within this document, the leading "snap." prefix is not usually shown, however within your code you will need to specify it.

For example:

```
save_nv_param( snap.NV_AES128_ENABLE_ID, snap.ENCRYPTION_TYPE_NONE )
```

Some constants have an additional prefix; for example the various "HOOK_xxx" constants are moved into a "hooks" group. Those prefixes will be shown in this section, but the leading "snap." portion must be added too.

For example:

```
set_hook(snap.hooks.HOOK_TRACEROUTE, trace_route_handler)
```

Constants used with encryption

ENCRYPTION_TYPE_NONE	Used to turn off encryption.
ENCRYPTION_TYPE_AES128	Used to enable AES-128 encryption.
ENCRYPTION_TYPE_BASIC	Used to enable basic SNAP encryption.

Constants used with close_serial(), open_serial(), HOOK_SERIAL_OPEN and HOOK_SERIAL_CLOSE

You will use the following constants as the serial_type parameter to the `close_serial()`, `open_serial()`, HOOK_SERIAL_OPEN, and HOOK_SERIAL_CLOSE handler functions.

SERIAL_TYPE_SNAPSTICK100	The SN132 SNAPstick, sometimes referred to as a "paddle board", or the SN163 demonstration board, sometimes referred to as a "bridge board". These are easily recognized, since they have no case (cover), and you can swap out the SNAP Engine on it for a different model. These have to be plugged into a USB port.
SERIAL_TYPE_SNAPSTICK200	The SS200 SNAP Stick, which is much smaller than the SN132, does have a plastic case, and does not accept plug-in SNAP Engines. (It is completely self-contained.) These have to be plugged into a USB port.
SERIAL_TYPE_RS232	"True" COM port, or a USB-serial cable.

Constants used with set_hook()

You will use the following constants as the hook parameter to the `set_hook()` function. They are described in more detail in the section SNAP Connect API Reference – HOOKS of this document, and so are merely listed here.

```
hooks.HOOK_SNAPCOM_OPENED
hooks.HOOK_SNAPCOM_CLOSED
hooks.HOOK_SERIAL_CLOSE
```

hooks.HOOK_SERIAL_OPEN

hooks.HOOK_RPC_SENT

hooks.HOOK_STDIN

hooks.HOOK_TRACEROUTE

hooks.HOOK_OTA_UPGRADE_COMPLETE

hooks.HOOK_OTA_UPGRADE_STATUS

Constants used with rpc_source_interface()

INTF_TYPE_UNKNOWN	(you should never see this)
INTF_TYPE_802154	For future use, as *SNAP Connect* currently relies on a "bridge" node to provide the radio
INTF_TYPE_SERIAL	RPC call came in over RS-232
INTF_TYPE_SILABS_USB	RPC call came in over a Silicon Labs USB interface chip (i.e., an SN132 or SN163)
INTF_TYPE_ETH	RPC call came in over TCP/IP
INTF_TYPE_SNAPSTICK	RPC call came in from an SS200 SNAPstick

Constants used with SPY Uploading

SNAPPY_PROGRESS_ERASE	Previous script has been erased
SNAPPY_PROGRESS_UPLOAD	"Chunk" of script accepted
SNAPPY_PROGRESS_COMPLETE	Upload completed successfully
SNAPPY_PROGRESS_ERROR	Upload failed
SNAPPY_PROGRESS_TIMEOUT	Node failed to respond
SNAPPY_PROGRESS_WRITE_ERROR	FLASH write failure
SNAPPY_PROGRESS_WRITE_REFUSED	Power too low to attempt
SNAPPY_PROGRESS_UNSUPPORTED	Node does not support script upload. For example, it is a SNAP Connect instance rather than an embedded node.

Constants used with firmware upgrades

OTA_PROGRESS_COMPLETE	Upgrade completed successfully
OTA_PROGRESS_ERROR	Upgrade failed
OTA_PROGRESS_CANCELED	Upgrade canceled
OTA_PROGRESS_TIMEOUT	Node failed to respond

OTA_PROGRESS_WRITE_ERROR	FLASH write failure
OTA_PROGRESS_WRITE_REFUSED	Power too low to attempt
OTA_PROGRESS_UNSUPPORTED	Node does not support over the air firmware upgrade

String Constants used with Logging

Python logging supports fine grained control of level (verbosity).

The levels that can be applied are DEBUG, INFO, WARNING, ERROR, and FATAL, where DEBUG is the most verbose, and FATAL is the least.

To change the log level globally, you would do something like:

```
log = logging.getLogger()
log.setLevel(logging.DEBUG)
```

To change the level on a per-module basis, you use the name of the module: “apy”, “SerialWrapper”, “snap”, or “snaplib”. For example:

```
snaplib_log = logging.getLogger('snaplib')
snaplib_log.setLevel(logging.ERROR)
```

Even finer grained control is possible, but you have to know the name (label) of the loggers you want to control. That is the purpose of this next list.

- “SerialWrapper”
  - “SerialWrapper.pySerialSocket”
- “snap”
  - “snap.AutoSaver”
  - “snap.Deferred”
  - “snap.PacketSink”
  - “snap.dispatchers”
  - “snap.listeners”
  - “snap.mesh”
  - “snap.SNAPtcpConnection”
  - “snap.SNAPtcpServer”
- “snaplib”
  - “snaplib.ComUtils”
  - “snaplib.EventCallbacks”
  - “snaplib.PacketQueue”
  - “snaplib.PacketSerialProtocol”
  - “snaplib.PySerialDriver”
  - “snaplib.RpcCodec”
  - “snaplib.TraceRouteCodec”
  - “snaplib.ScriptsManager”
  - “snaplib.SerialConnectionManager”
  - “snaplib.SnappyUploader”

For example:

```
snaplib_log = logging.getLogger('snaplib.RpcCodec')
snaplib_log.setLevel(logging.INFO)
```

7. *SNAP Connect* API Reference – NV Parameters

Embedded SNAP Nodes keep configuration parameters in physical Non-Volatile (NV) memory.

SNAP Connect emulates this type of configuration repository using a standard Python **pickle** file named "`nvparams.dat`".

The following non-volatile parameters are available through the `save_nv_param()` and `load_nv_param()` API functions.

NOTE – Unlike in embedded SNAP nodes, *SNAP Connect* NV Parameter changes take effect immediately (no reboot required).

Here are all of the System (Reserved) NV Parameters (sorted by numeric ID) that apply to *SNAP Connect*, and what they do. You can use these same constants when accessing NV parameters on an embedded node from a *SNAP Connect* script, even if the parameter has no meaning in the *SNAP Connect* context (such as querying for a node's radio link quality threshold).

NOTE – Embedded SNAP Nodes use these same NV parameters, plus **many more.** Refer to the **SNAP Reference Manual** for more information.

NOTE – You can also define your own NV Parameters (in the range 128-254) which your script can access and modify, just like the system NV Parameters.

ID 0-1 – Reserved for Synapse Use, not used by *SNAP Connect*

ID 2 – MAC Address, not used by *SNAP Connect*

Enumeration = `snap.NV_MAC_ADDR_ID`

ID 3 – Network ID, not used by *SNAP Connect*

Enumeration = `snap.NV_NETWORK_ID`

ID 4 – Channel, not used by *SNAP Connect*

Enumeration = `snap.NV_CHANNEL_ID`

ID 5 – Multi-cast Processed Groups

Enumeration = `snap.NV_GROUP_INTEREST_MASK_ID`

This is a 16-bit field controlling which multi-cast groups the node will respond to. It is a bit mask, with each bit representing one of 16 possible multi-cast groups. For example, the 0x0001 bit represents the default group, or "broadcast group." (Any node not a member of this group will not respond to route requests, and thus will not participate in mesh routing for other nodes.)

One way to think of groups is as "logical sub-channels" or as "subnets." By assigning different nodes to different groups, you can further subdivide your network.

For example, Portal could multi-cast a "sleep" command to group 0x0002, and ***only*** nodes *with that bit set* in their **Multi-cast Processed Groups** field would go to sleep. (This means nodes with their group values set to 0x0002, 0x0003, 0x0006, 0x0007, 0x000A, 0x000B, 0x000E, 0x000F, 0x0012, etc., would respond.) Note that a single node can belong to any (or even all) of the 16 groups.

Group membership does not affect how a node responds to a direct RPC call. It only affects multi-cast requests.

ID 6 – Multi-cast Forwarded Groups

Enumeration = `snap.NV_GROUP_FORWARDING_MASK_ID`

This is a separate 16-bit field controlling which multi-cast groups will be *re-transmitted* (forwarded) by the node. It is a bit mask, with each bit representing one of 16 possible multi-cast groups. For example, the 0x0001 bit represents the default group, or "broadcast group."

By default, all nodes process and forward group 1 (*broadcast*) packets.

Please note that the Multi-cast Processed Groups and Multi-cast Forwarded Groups fields are independent of each other. A node could be configured to forward a group, process a group, or both. It can process groups it does not forward, or vice versa.

NOTE – Every interface opened in a *SNAP Connect* application has a `forward_groups=None` parameter available when you open (or listen for) the interface. This parameter restricts which multicast messages will be forwarded through that interface from the *SNAP Connect* instance. A default value of `None` indicates that the interface will only allow multicast packets for groups that the *SNAP Connect* instance itself would forward.

The hazard here is that if you do not explicitly set the `forward_groups` parameter when you open a connection, *SNAP Connect* will only be able to make multicast requests of nodes in groups that it would normally forward. In other words, if you want *SNAP Connect* to be able to initiate multicast requests to a group that it would not normally forward, you must explicitly set that group as a forwarding group when you open the interface.

NOTE – If you set your bridge node to not forward multi-cast commands, any *Portal* or *SNAP Connect* attached to that bridge will not be able to multi-cast to the rest of your network.

ID 7 – Manufacturing Date, not used by *SNAP Connect*

Enumeration = `snap.NV_MFG_DATE_ID`

ID 8 – Device Name

Enumeration = `snap.NV_DEVICE_NAME_ID`

This NV Parameter lets you choose a name for the node. If this parameter is set to None, then the node name will default to "SNAPcom". You do not have to give your nodes explicit names.

NOTE – It is invalid to put embedded spaces in your Device Name. "My Node" is not a legal name, while "My_Node" is.

ID 9 – Last System Error, not used by *SNAP Connect*

Enumeration = `snap.NV_SYSTEM_ERROR_ID`

ID 10 – Device Type, not used by *SNAP Connect*

Enumeration = `snap.NV_DEVICE_TYPE_ID`

ID 11 – Feature Bits

Enumeration = `snap.NV_FEATURE_BITS_ID`

These control some miscellaneous hardware settings on embedded SNAP Nodes. The only Feature Bits that apply to a *SNAP Connect* instance are:

Bit 8 (0b0000,0001,0000,0000 0x0100) – Enable second RPC CRC
Bit 10 (0b0000,0100,0000,0000 0x0400) – Enable packet CRC

The second CRC bit (0x100) enables a second CRC packet integrity check on platforms that support it. Setting this bit tells the SNAP node to send a second cyclical redundancy check (using a different CRC algorithm) on each RPC or multicast packet, and require this second CRC on any such packet it receives. This reduces the available data payload by two bytes (to 106 bytes for an RPC message, or 109 bytes for a multicast message), but provides an additional level of protection against receiving (and potentially acting upon) a corrupted packet. The CRC that has always been a part of SNAP packets means that there is a one in 65,536 chance that a corrupted packet might get interpreted as valid. The second CRC should reduce this to a less than a one in four billion chance.

If you set this bit for the second RPC CRC, you must set it in all nodes in your network, and enable the feature in your Portal preferences or as a feature bit in your *SNAP Connect* NV parameters.

A node that does not have this parameter set will be able to hear and act on messages from a node that does have it set, but will not be able to communicate back to that node. Not all platforms support this second CRC. Refer to each platform's details in the SNAP Reference Manual to determine whether this capability is available.

The packet CRC bit (0x400) enables *SNAP Connect* to communicate with a network where packet CRC is enabled. This applies a second CRC to *every* packet instead of just RPC (and multicast RPC) packets. Enabling both packet CRC and second RPC CRC will result in RPC packets using three CRC values when sent over the air.

Unlike the second RPC CRC, the packet CRC is not used over packet serial connections. Packets traveling between the bridge node and *SNAP Connect* will not have the extra CRC applied. However, because these packets may end up over the air, this feature needs to be enabled in *SNAP Connect* to ensure your packets do not exceed the maximum size when this feature is enabled.

If you set this bit for the packet CRC, you should set it in all nodes in your network, and enable the feature in your Portal preferences or as a feature bit in your *SNAP Connect* NV parameters.

Not all platforms support this packet CRC. Refer to each platform's details in the SNAP Reference Manual to determine whether this capability is available.

Binary notations are provided here for clarity. You should specify the parameter value using the appropriate hexadecimal notation. For example, 0x041F corresponds to 0b0000,0100,0001,1111.

ID 12 – Default UART, not used by *SNAP Connect*

Enumeration = `snap.NV_DEFAULT_UART_ID`

ID 13 – Buffering Timeout, not used by *SNAP Connect*

Enumeration = `snap.NV_UART_DM_TIMEOUT_ID`

ID 14 – Buffering Threshold, not used by *SNAP Connect*

Enumeration = `snap.NV_UART_DM_THRESHOLD_ID`

ID 15 – Inter-character Timeout, not used by *SNAP Connect*

Enumeration = `snap.NV_UART_DM_INTERCHAR_ID`

ID 16 – Carrier Sense, not used by *SNAP Connect*

Enumeration = `snap.NV_CARRIER_SENSE_ID`

ID 17 – Collision Detect, not used by *SNAP Connect*

Enumeration = `snap.NV_COLLISION_DETECT_ID`

ID 18 – Collision Avoidance, not used by *SNAP Connect*

Enumeration = `snap.NV_COLLISION_AVOIDANCE_ID`

ID 19 – Unicast Retries

Enumeration = `snap.NV_SNAP_MAX_RETRIES_ID`

This lets you control the number of unicast transmit attempts. This parameter defaults to 8.

This parameter refers to the total number of attempts that will be made to get an acknowledgement back on a unicast transmission to another node.

In some applications, there are time constraints on the "useful lifetime" of a packet. In other words, if the packet has not been successfully transferred by a certain point in time, it is no longer useful. In these situations, the extra retries are not helpful – the application will have already "given up" by the time the packet finally gets through.

By lowering this value from its default value of 8, you can tell SNAP to "give up" sooner. A value of 0 is treated the same as a value of 1 – a packet gets at least one chance to be delivered no matter what.

If your connection link quality is low and it is important that every packet get through, a higher value here may help. However it may be appropriate to reevaluate your network setup to determine if it would be better to change the number of nodes in your network to either add more nodes to the mesh to forward requests, or reduce the number of nodes broadcasting to cut down on packet collisions.

ID 20 – Mesh Routing Maximum Timeout

Enumeration = `snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID`

This indicates the maximum time (in milliseconds) a route can "live." This defaults to 0xEA60 (60,000), or one minute.

Discovered mesh routes timeout after a configurable period of inactivity (see #23), but this timeout sets an upper limit on how long a route will be used, even if it is being used heavily. By forcing routes to be rediscovered periodically, the nodes will have an opportunity to discover a more efficient route if one becomes available..

Note that you *can* set this timeout to zero (which will disable it) if you know for certain that your nodes are stationary, or have some other reason for needing to avoid periodic route re-discovery.

You can use `get info(15)` to determine the number of currently active (not timed out) routes.

ID 21 – Mesh Routing Minimum Timeout

Enumeration = `snap.NV_MESH_ROUTE_AGE_MIN_TIMEOUT_ID`

This is the minimum time (in milliseconds) a route will be kept. This defaults to 1000, or one second. Note that in a busy mesh environment, routes may fall out of the route table before this time period has elapsed if the route table becomes full and another route is discovered.

ID 22 – Mesh Routing New Timeout

Enumeration = `snap.NV_MESH_ROUTE_NEW_TIMEOUT_ID`

This is the grace period (in milliseconds) that a newly discovered route will be kept, even if it is never actually used. This defaults to 5000, or five seconds.

ID 23 – Mesh Routing Used Timeout

Enumeration = `snap.NV_MESH_ROUTE_USED_TIMEOUT_ID`

This is how many additional milliseconds of "life" a route gets whenever it is used. This defaults to 5000, or five seconds.

Every time a known route gets used, its timeout gets reset to this parameter. This prevents active routes from timing out as often, but allows inactive routes to go away sooner. See also Parameter #20, which takes precedence over this timeout.

ID 24 – Mesh Routing Delete Timeout

Enumeration = `snap.NV_MESH_ROUTE_DELETE_TIMEOUT_ID`

This timeout (in milliseconds) controls how long "expired" routes are kept around for bookkeeping purposes. This defaults to 10000, or 10 seconds.

ID 25 – Mesh Routing RREQ Retries

Enumeration = `snap.NV_MESH_RREQ_TRIES_ID`

This parameter controls the total number of retries that will be made when attempting to "discover" a route (a multi-hop path) over the mesh. This defaults to 3.

ID 26 – Mesh Routing RREQ Wait Time

Enumeration = `snap.NV_MESH_RREQ_WAIT_TIME_ID`

This parameter (in milliseconds) controls how long a node will wait for a response to a **Route Request (RREQ)** before trying again. This defaults to 500, or a half second.

Not that subsequent retries use longer and longer timeouts (the timeout is doubled each time). This allows nodes from further and further away time to respond to the RREQ packet.

ID 27 – Mesh Routing Initial Hop Limit

Enumeration = `snap.NV_MESH_INITIAL_HOPLIMIT_ID`

This parameter controls how far the initial "discovery broadcast" message is propagated across the mesh.

If your nodes are geographically distributed such that they are always more than 1 hop away from their logical peers, then you can increase this parameter. Consequently, if most of your nodes are within direct radio range of each other, having this parameter at the default setting of 2 will use less radio bandwidth.

SNAP Connect does not support disabling mesh discovery by setting this value to 0. A value of 0 will be treated as if you had set the initial hop limit to the default value of 2.

This parameter should remain less than or equal to the next parameter, ***Mesh Routing Maximum Hop Limit***.

ID 28 – Mesh Routing Maximum Hop Limit

Enumeration = `snap.NV_MESH_MAX_HOPLIMIT_ID`

To cut down on needless broadcast traffic during mesh networking operation (thus saving both power and bandwidth), you can choose to lower this value to the maximum number of physical hops across your network. The default value is 5.

NOTE – if your network is larger than 5 hops, you will need to raise this parameter.

ID 29 – Mesh Sequence Number

Enumeration = `snap.NV_MESH_SEQUENCE_NUMBER_ID`

Reserved for future use.

ID 30 – Mesh Override

Enumeration = `snap.NV_MESH_OVERRIDE_ID`

This is used to limit a node's level of participation within the mesh network.

When set to the default value of 0, the node will fully participate in the mesh networking. This means that not only will it make use of mesh routing, but it will also "volunteer" to route packets for other nodes.

Setting this value to 1 will cause the node to stop volunteering to route packets for other nodes. It will still freely use the entire mesh for its own purposes (subject to other nodes' willingness to route packets for it).

This feature was added to better support nodes that spend most of their time "sleeping." If a node is likely to spend most of its time asleep, there may be no point in it becoming part of routes *for other nodes* while it is (briefly) awake.

This can also be useful if some nodes are externally powered, while others are battery-powered. Assuming sufficient radio coverage (all the externally powered nodes can "hear" all of the other nodes), then the **Mesh Override** can be set to 1 in the battery powered nodes, extending their battery life at the expense of reducing the "redundancy" in the overall mesh network.

NOTE – Enabling this feature on your bridge node means Portal will no longer be able to communicate with the rest of your network, regardless of how everything else is configured. No nodes in your network (except for your bridge node) will be able to receive commands or information from Portal or send commands or information to Portal.

ID 31 – Mesh Routing LQ Threshold, not used by *SNAP Connect*

Enumeration = `snap.NV_MESH_PENALTY_LQ_THRESHOLD_ID`

ID 32 – Mesh Rejection LQ Threshold, not used by *SNAP Connect*

Enumeration = `snap.NV_MESH_REJECTION_LQ_THRESHOLD_ID`

ID 33 – Noise Floor, not used by *SNAP Connect*

Enumeration = `snap.NV_CS_CD_LEVEL_ID`

ID 34-38 – Reserved for Future Use, not used by *SNAP Connect*

ID 39 – Radio LQ Threshold, not used by *SNAP Connect*

Enumeration = `snap.NV_SNAP_LQ_THRESHOLD_ID`

ID 40 – SNAPpy CRC, not used by *SNAP Connect*

Enumeration = `snap.NV_SNAPPY_CRC_ID`

ID 41 – Platform, not used by *SNAP Connect*

Enumeration = `snap.NV_SYS_PLATFORM_ID`

ID 42-49 – Reserved for Future Use

ID 50 – Enable Encryption

Enumeration = `snap.NV_AES128_ENABLE_ID`

Control whether encryption is enabled, and what type of encryption is in use for firmware that supports multiple forms. The options for this field are:

0 = Use no encryption. (This is the default setting.)

1 = Use AES-128 encryption if supported.

2 = Use Basic encryption.

If you set this to a value that indicates encryption should be used, but either an invalid encryption key is specified (in NV Parameter #51), or your Python environment does not support the encryption mode specified, your transmissions will not be encrypted.

SNAP versions before 2.4 did not include the option for Basic encryption, and nodes upgraded from those firmware versions may contain False or True for this parameter. Those values correspond to 0 and 1 and will continue to function correctly. Basic encryption is not as secure as AES-128 encryption, but it is available in all nodes starting with release 2.4.

If encryption is enabled and a valid encryption key is specified, all communication from the node will be encrypted, whether it is sent over the air or over a serial connection. Likewise, the node will expect that all communication to it is encrypted, and will be unable to respond to unencrypted requests from other nodes. If you have a node that you cannot contact because of a forgotten encryption key, you will have to reset the factory parameters on the node to reestablish contact with it.

Enabling AES-128 encryption in *SNAP Connect* requires that you have the third-party PyCrypto Python library installed. You can acquire the library from http://pycrypto.org/

ID 51 – Encryption Key

Enumeration = `snap.NV_AES128_KEY_ID`

The encryption key used by either AES-128 encryption or Basic encryption, if enabled. This NV Parameter is a string with default value of “”. If you are enabling encryption, you must specify an encryption key. Your encryption key should be complex and difficult to guess, and it should avoid repeated characters when possible.

An encryption key must be exactly 16 bytes (128 bits) long to be valid. This parameter has no effect unless NV parameter #50 is also set to enable encryption.

If NV parameter #50 is set for AES-128 encryption and parameter 51 has a valid encryption key, *SNAP Connect* will fail with an exception if you do not have the PyCrypto Python library installed.

ID 52 – Lockdown

Enumeration = `snap.NV_LOCKDOWN_FLAGS_ID`

If this parameter is 0 (or never set at all), access is unrestricted. You can freely change NV Parameters (even remotely).

If you set this parameter to 2, then the system enters a "lockdown" mode where remote NV Parameter changes are disallowed.

Values other than 0 or 2 are reserved for future use, and should not be used on a *SNAP Connect* node. (The value 1 is valid on embedded nodes, preventing over-the-air changes to the node's SNAPpy script.)

ID 53 – Maximum Loyalty, not used by *SNAP Connect*

Enumeration = `snap.NV_MAX_LOYALTY_ID`

ID 54 – Reserved for Future Use

ID 55 – Alternate Encryption Key (*SNAP Connect* only)

Enumeration = `snap.NV_ALTERNATE_KEY_ID`

This NV parameter is like NV #51 in form and function. It represents a second ("alternate") encryption key that can be used in special circumstances.

For example, you could be running a fully encrypted network using key "encryption_key_1" when a new node is added with encryption key "KEY2(ENCRYPTION)". By saving the second key in NV #55, and proper use of the `add rpc func()`, `rpc use alternate key()`, and `rpc alternate key used()` functions you can perform **multicast** interactions with the new node, while still continuing to interact fully (both multicast and unicast) with the old nodes. This gives you a way to reprogram the new node to switch over to using the first key without having to pause communications to the rest of the network.

ID 56-59 – Reserved for Future Use

ID 60 – Last Version Booted *(Deprecated)*, not used by *SNAP Connect*

Enumeration = `snap.NV_LAST_VERSION_BOOTED_ID`

ID 61 – Reboots Remaining, not used by *SNAP Connect*

Enumeration = `snap.NV_REBOOTS_REMAINING_ID`

ID 62 – Reserved for Future Use

ID 63 – Alternate Radio Trim value, not used by *SNAP Connect*

Enumeration = `snap.NV_ALT_RADIO_TRIM_ID`

ID 64 – Vendor-Specific Settings, not used by *SNAP Connect*

Enumeration = `snap.NV_VENDOR_SETTINGS_ID`

ID 65 – Clock Regulator, not used by *SNAP Connect*

Enumeration = `snap.NV_CLOCK_REGULATOR_ID`

ID 66 – Radio Calibration Data, not used by *SNAP Connect*

Enumeration = `snap.NV_RADIO_CALIBRATION_ID`

ID 67-69 – Reserved for Future Use

ID 70 – Transmit Power Limit, not used by *SNAP Connect*

Enumeration = `snap.NV_TX_POWER_LIMIT_ID`

ID 71-127 – Reserved for Future Use

ID 128-254 – Available for User Definition

These are user-defined NV Parameters, and can be used for whatever purpose you choose, just as with embedded SNAP Nodes. Unlike embedded SNAP nodes, however, *SNAP Connect* lets you store any pickleable object in an NV parameter, rather than just a string, Boolean, integer, or None.

ID 255 – Reserved for Synapse Use, not used by *SNAP Connect*

8. *SNAP Connect* API Reference – Exceptions

When invoking *SNAP Connect* functions, your program should be prepared to "catch" any Python exceptions that are "thrown" (raised).

The following lists the possible Python exceptions that can be thrown by the *SNAP Connect* libraries, and some possible causes (exception text messages).

NOTE – Obvious exceptions like **Exception**, **KeyboardInterrupt**, and **SystemExit** are not shown.

Exceptions that can be thrown from the "apy" package

The monotime module in the apy package can throw the following exception

OSError

The PostThread module in the apy package can throw the following exceptions

RuntimeError

Exceptions that can be thrown from the "serialwrapper" package

The ComUtil module in the serialwrapper package can throw the following exceptions

NoMorePortsError

- "No more ports to scan"

Exception

- "Did not receive expected response"
- "Unknown probe version"

The ftd2xxserialutil module in the serialwrapper package can throw the following exceptions

Ftd2xxSerialException

- "Cannot set number of devices"

ValueError

- "Serial port MUST have enabled timeout for this function!"
- "Not a valid port: …"
- "Not a valid baudrate: …"
- "Not a valid byte size: …"
- "Not a valid parity: …"
- "Not a valid stopbit size: …"
- "Not a valid timeout: …"

The ftd2xxserialwin32 module in the serialwrapper package can throw the following exceptions

Ftd2xxSerialException

- "Port must be configured before it can be used"
- "Could not find devices to open: …"
- "Could not find port: …"
- "Could not open device: …"

- “Could not set latency: …”
- “Can only operate on an open port”
- “Could not set timeouts: …”
- “Could not set baudrate: …”
- “Could not set break: …”
- “Could not reset break: …”
- “Could not get CTS state: …”
- “Could not get DSR state: …”
- “Could not get RI state: …”
- “Could not get DCD state: …”
- “Could not set stopbits, parity, and/or bits per word: …”
- “Could not set flow control: …”
- “Cannot configure port, some setting was wrong…”
- “An error occurred while checking rx queue: …”
- “An error occurred while reading: …”
- “An error occurred while writing: …”
- “An error occurred while flushing the input buffer: …”
- “An error occurred while setting RTS: …”
- “An error occurred while clearing RTS: …”

The PyserialDriver module in the serialwrapper package can throw the following exceptions

Exception

- “Buffers don’t match”

Serial.SerialException

- “WriteFile failed…”
- “write failed: …”

PortNotOpenError

TypeError

- “Expected str or bytearray, got …”
- “Unknown serial driver type”
- “Unsupported output type”

SerialOpenException

- “SNAP USB devices are not currently supported on this platform”
- “An error occurred while setting up the serial port: …”

The usbxserialutil module in the serialwrapper package can throw the following exceptions

ValueError

- “Serial port MUST have enabled timeout for this function!”
- “Not a valid port: …”
- “Not a valid baudrate: …”
- “Not a valid byte size: …”
- “Not a valid parity: …”

- “Not a valid stopbit size: …”

“Not a valid timeout: …”

UsbxSerialException

- “Cannot set number of devices”

The usbxserialwin32 module in the serialwrapper package can throw the following exceptions

portNotOpenError

UsbxSerialException

- “Port must be configured before it can be used”
- “Unable to verify device PID: …”
- “USB device was not found to be a SNAP USB device”
- “Could not open device: …”
- “Can only operate on an open port”
- “Could not set timeouts: …”
- “Could not set baud rate: …”
- “Could not set stop bits, parity, and/or bits per word: …”
- “Could not set flow control: …”
- “Cannot configure port, some setting was wrong: …”
- “Could not get CTS state…”
- “An error occurred while checking rx queue: …”
- “An error occurred while reading: …”
- “Write time out occurred and there are no USB devices”
- “A system error occurred while writing: …”
- “An error occurred while flushing the input buffer: …”
- “An error occurred while flushing the output buffer: …”

Exceptions that can be thrown from the “snapconnect” package

The auth_digest module in the snapconnect package can throw the following exceptions

cherrypy.HTTPError

- “Bad Request…”
- “You are not authorized to access that resource”

ValueError

- “Authorization scheme is not Digest”
- “Unsupported value for algorithm…”
- “Not all required parameters are present”
- “Unsupported value for qop: …”
- “If qop is sent then cnonce and nc MUST be present”
- “If qop is not sent, neither cnonce nor nc can be present”
- “Unrecognized value for qop: …”

The dispatchers module in the snapconnect package can throw the following exception

TypeError

- “Unknown descriptor type”

The LicenseManager module in the snapconnect package can throw the following exception

LicenseError

- “… license file has expired”
- “Your license is invalid…”
- “Unable to load license file…”

The listeners module in the snapconnect package can throw the following exception

ValueError

- “Unknown serial type”

The snap module in the snapconnect package can throw the following exceptions

ImportError

- “Unable to find PyCrypto library required for AES support”

IOError

- “Unable to load NV params file: …”

RuntimeError

- “Non-licensed address provided”
- “Invalid license found”
- “Unable to determine callable functions”
- “Tornado ioloop not supported”
- “You must be accepting TCP connections to call accept_sniffer()”

TypeError

- “Unknown Hook Type”
- “Invalid callback”
- “Invalid connection parameter provided”

ValueError

- “Unknown encryption type specified: …”
- “auth_info is not callable”
- “Serial interface type must be an integer”
- “Unsupported serial interface type”

The snaptcp module in the snapconnect package can throw the following exceptions

socket.error

- “Did not receive a valid lookup result”

ValueError

- “SSL support was not found”

Exceptions that can be thrown from the “snaplib” package

The DataModeCodec module in the snaplib package can throw the following exception

DataModeEncodeError

- “Packet is greater than 255 bytes”
- “The packet is too large to encode …”

The MeshCodec module in the snaplib package can throw the following exception

MeshError

- “encode function does not yet support message type …”
- “Packet is greater than 255 bytes”
- “Mesh Routing message too large to encode …”

The RpcCodec module in the snaplib package can throw the following exceptions

RpcError

- “Do not receive a list of arguments”
- “Source Address must be 3 bytes”
- “Source Address must be between \xFF\xFF\xFF and \x00\x00\x00”
- “No source address was set”
- “No destination address/group was set”
- “dmcast_dstAddrs length must be a multiple of 3 (including 0)”
- “Original TTL for multicast must be greater than 0 and less than 256”
- “Delay Factor for directed multicast must be between 0 and 255”
- “The multicast group must be 2 bytes”
- “Destination multicast group cannot be all zeros”
- “TTL for multicast must be greater than 0 and less than 256”
- “Packet cannot be a Directed Multicast without being a Multicast Packet first”
- “The destination address must be 3 bytes”
- “Int out of range”
- “Function name cannot be None”
- “Max string length is 255”
- “Unsupported DataType: …”
- “Packet is greater than 255 bytes”
- “Function/Args too large to encode …”

TypeError

- “Unsupported type …”
- “String length is greater than 255”
- “Integer is out of range”

WrongArgCount

The SnappyUploader module in the snaplib package can throw the following exception

AlreadyInProgressError

- “There is currently a SNAPpy upload already in progress”

The TraceRouteCodec module in the snaplib package can throw the following exception

TraceRouteEncodeError

- “The data is too large to encode”
- “Packet is greater than 255 bytes”

9. *SNAP Connect* API Reference – HOOKS

SNAP Hooks are events that can occur at runtime. This section lists the supported hooks, their meanings, their triggers, and their (callback) parameters.

Refer also to the description of the `set hook()` function earlier in this manual.

Here the focus is on the individual hooks and their handlers.

HOOK_SNAPCOM_OPENED – SNAP COMmunications established

Meaning: SNAP COMmunications have been established over a TCP/IP link.

Triggered by:

Establishment of an inbound or outbound TCP/IP connection with another instance of *SNAP Connect* (possibly embedded inside of another application like *Portal*).

This means the root cause of the HOOK_SNAPCOM_OPENED was either:

1) a call to `accept tcp()` that allowed/enabled incoming TCP/IP connections; or
2) a call to `connect tcp()` that created an outgoing TCP/IP connection.

Required callback signature:

```
some_function(connection_info, snap_address)
```

where the parameters are:

`connection_info` : this is a Python tuple containing **two pieces** of information:

`connection_info[0]` is the IP address of the other *SNAP Connect* instance.

`connection_info[1]` is the TCP/IP port of the connection.

`snap_address` : this is a Python string containing the three-byte SNAP Address of the other node.

Hint – if you want to know your *own* SNAP Address, try the `local addr()` function.

HOOK_SNAPCOM_CLOSED – SNAP COMmunications ended

Meaning: SNAP COMmunications have ended (or failed to ever get established) over a TCP/IP link.

Triggered by:

Shutdown of an established TCP/IP connection, or failure to establish a connection in the first place (for example, calling `connect tcp()` with the IP address of a computer that is **not** running *SNAP Connect*).

Required callback signature:

```
some_function(connection_info, snap_address)
```

where the parameters vary slightly depending on the cause:

Scenario 1 – connection could not be established

`connection_info` : this is a Python tuple containing **two pieces** of information:

`connection_info[0]` is the IP address of the other computer.

`connection_info[1]` is the TCP/IP port of the initial connection attempt.

`snap_address` : in this scenario, *snap_address* is None because there **was no** SNAP Node at the other end of the attempted connection.

Scenario 2 – an established connection was later shut down

`connection_info` : this is a Python tuple containing **two pieces** of information:

`connection_info[0]` is the IP address of the other *SNAP Connect* instance.

`connection_info[1]` is the TCP/IP port of the connection.

`snap_address` : this is a Python string containing the three-byte SNAP Address of the other node.

In other words, the parameters for a valid connection closing are the same as those when it was initially opened. (See `HOOK_SNAPCOM_OPENED`).

HOOK_SERIAL_OPEN – Serial Communications started

Meaning: Communications over a specific serial (USB or RS-232) interface have started.

Triggered by:

This event will occur after every successful call to `open serial()`. After `open serial()` is called, *SNAP Connect* will attempt to retrieve the address of the device attached to the serial port. This event will trigger after the address has been retrieved or the retrieval attempt has timed out.

Because the serial connection is open immediately after the call to `open serial()`, it is possible that packets may be received on the interface before the node address can be determined and the hook function is called.

Required callback signature:

```
some_function(serial_type, port, addr)
```

where the parameters are:

`serial_type` : The type of the serial port, one of:

`SERIAL_TYPE_RS232`

`SERIAL_TYPE_SNAPSTICK100`

`SERIAL_TYPE_SNAPSTICK200`

`port` : The port **of that particular type** that opened, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.) On Linux, port will typically be a string, for example `"/dev/ttys1"`.

`addr` : The address of the SNAP device attached to this serial port, or **None** if the address could not be retrieved for any reason.

NOTE – These parameters are the same ones used in the initial `open serial()` call, and in any subsequent `close serial()` call.

HOOK_SERIAL_CLOSE – Serial Communications ended

Meaning: Communications over a specific serial (USB or RS-232) interface have ended.

Triggered by:

A HOOK_SERIAL_CLOSE event can be triggered by a call to close serial(), or (in the case of some removable USB devices) by the physical removal of the device from the computer *SNAP Connect* is running on.

This event can only occur after a successful call to open serial().

Required callback signature:

```
some_function(serial_type, port)
```

where the parameters are:

serial_type : The type of the serial port, one of:

SERIAL_TYPE_RS232

SERIAL_TYPE_SNAPSTICK100

SERIAL_TYPE_SNAPSTICK200

port : The port **of that particular type** that closed, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example "/dev/ttys1 ".

NOTE – These parameters are the same ones used in the initial open serial() call, and in any subsequent close serial() call.

HOOK_RPC_SENT – RPC packet processing complete

Meaning: A packet created previously by a call to rpc(), mcast rpc(), or dmcast rpc() has been sent, or given up on (all retries were exhausted while attempting to send the packet).

Triggered by:

This event fires after any RPC packet is sent, regardless of whether it was sent by your application code, or automatically by *SNAP Connect* behind the scenes.

Required callback signature:

```
some_function(packet_identifier, transmit_success)
```

where the parameters are:

packet_identifier : The identifier for which packet was sent.

NOTE – This is the same identifier that was originally returned by the call to rpc(), mcast rpc(), or dmcast rpc(). For compatibility with embedded SNAP nodes, you can also recover this identifier immediately after sending an RPC using get info(9).

transmit_success : True if SNAP Connect believes the packet has been transmitted successfully or False if SNAP Connect was unable to send the packet. A successful transmit **does not** indicate that the packet was able to reach the destination.

NOTE – Receipt of a HOOK_RPC_SENT does not mean the packet made it all the way to the other node. This **is not** an end-to-end acknowledgement! If transmit_success is False, the packet has not been transmitted; but if it is True, that is no guarantee the destination node networked through the interface actually received the packet.

SNAP has no provisions for end-to-end acknowledgement at the protocol layer. Any such functionality must be implemented in your application. For example, have a node send a return RPC in response to a request RPC. Only when the originating node receives the response RPC can it be certain that the original request made it through.

HOOK_STDIN – Data Mode data received

Meaning: A packet containing user data has been received. This could either be unicast data addressed specifically to this node, or it could be multicast data sent to any nodes within specified multicast group[s].

Triggered by:

This event is ultimately triggered by some other node invoking `data_mode()` or `mcast_data_mode()`, if it is a *SNAP Connect* application.

Alternatively, an embedded SNAP Node (such as an *RF100*) could be forwarding data (from a serial port for example) by use of the `ucastSerial()` or `mcastSerial()` functions (refer to the **SNAP Reference Manual**).

Required callback signature:

```
some_function(data)
```

where the parameter is:

`data` : The actual data (payload) of the DATA MODE packet.

Hint – if you need to know who the data came from, try the `rpc_source_addr()` function.

HOOK_TRACEROUTE – Trace Route data received

Meaning: A Trace Route has been successfully completed.

Triggered by:

This event is triggered by invoking the `traceroute()` function for a node that actually is reachable and that can respond.

Required callback signature:

```
some_function(node_addr, round_trip_time, hops)
```

where the parameters are:

`node_addr` : A Python string containing the three-byte SNAP Address of the node that was "traced".

`round_trip_time` : the round trip time for the trace route packet, after it made it past the initial hop.

NOTE - The first hop is not counted in Trace Routes because when the feature was originally implemented it was decided to only track the over-the-air timing. Bottom line – the bridge node component of your `round_trip_time` will be reported as 0 milliseconds. For TCP connections, this could significantly under-report the speed of that first hop.

`hops` : This is a Python list of tuples, with one list entry for every "hop" that the trace route made on its journey to and from the target node.

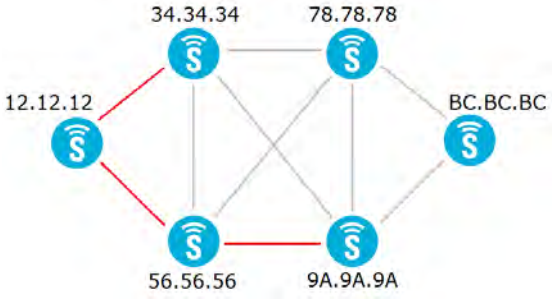
For example, a Trace Route to your directly connected bridge node will have two hops – one for the hop from *SNAP Connect* to the bridge node, and a second hop back from the bridge node to *SNAP Connect*.

Each Python tuple within the hops list will be made up of two values:

hop[0] will be the SNAP Address that the Trace Route packet was received from.

hop[1] will be the Link Quality at which the Trace Route was received, ***if*** it was received over a radio link. (Serial links and TCP/IP links have no true "link quality" in SNAP, and will always be reported as having a LQ of 0.) The link quality is reported in (negative) dBm, so lower values represent stronger signals. The theoretical range for the link quality is 0-127.

As an example of how a traceroute hook's return data might appear, consider this network where each red line indicates a serial (or TCP/IP) communication route, and each gray line indicates an over-the-air communication route between two nodes. If node 12.12.12 were to send a traceroute to node BC.BC.BC, then, depending on the routes determined by the nodes during route discovery, the hops parameter returned to the hooked function could look like this:




```
[('\x12\x12\x12', 0), ('\x56\x56\x56', 0), ('\x9A\x9A\x9A', 21), ('\xBC\xBC\xBC',
17), ('\x78\x78\x78', 19), ('\x34\x34\x34', 0)]
```

Of course, that is only one possibility, depending on factors such as signal strength and which nodes were responsive when routes were established. More commonly the return route would trace through the same nodes used on the way out, but that doesn't have to be the case.

For an example of how Trace Route results can be displayed, refer to the *Portal* user interface.

HOOK_TOPOLOGY_POLL_RESULTS – Topology poll results available

Meaning: A previously initiated topology poll has results that are available

Triggered by:

This event follows a call to the `topology_poll()` function. The event is triggered when the data of the poll has been gathered and is now available for user inspection.

Required callback signature:

`some handler_function(results_list, final_report)`

where the parameters are:

`results_list` : A list of neighbor Snap addresses.

`final_report`: Boolean – are we done polling?

NOTE - Due to packet size limitations, multiple invocations of the HOOK_TOPOLOGY_POLL_RESULTS handler may be necessary to capture ALL of the neighboring node addresses. On all but the LAST invocation of the HOOK_TOPOLOGY_POLL_RESULTS handler, the final_report parameter will be false. When all addresses have been gathered, this last invocation will result in a True value for the final_report parameter. If you are simply printing out the results as they come in, this probably does not matter, but if you are doing something like counting the total number of neighbor nodes, be sure to wait until you have received ALL of the results (final_report is True). It is worth noting here that it is valid for the last invocation to have an empty results_list (`results_list = []`).

HOOK_OTA_UPGRADE_COMPLETE – Over the air upgrade complete

Meaning: An over-the-air firmware upgrade has completed.

Triggered by:

This event follows a call to the `upgrade_firmware()` function. The event is triggered when an upgrade completes successfully or an error occurs that halts the upgrades progress.

Required callback signature:

```
some_function(upgrade_addr, status_code, message)
```

where the parameters are:

`upgrade_addr` : The address of the node for which the upgrade has completed.

`status_code` : The result of the upgrade. See the Constants used with firmware upgrades section for more details.

`message` : A human-readable string describing the cause of an error, or None if the upgrade was successful.

HOOK_OTA_UPGRADE_STATUS – Over the air upgrade status report

Meaning: An over-the-air firmware upgrade status has advanced.

Triggered by:

This event follows a call to the `upgrade_firmware()` function. The event is triggered every time progress is made in an over the air upgrade and will be called many times for every upgrade started and not immediately stopped.

Required callback signature:

```
some_function(upgrade_addr, percent_complete)
```

where the parameters are:

`upgrade_addr` : The address of the node receiving the upgrade.

`percent_complete` : An estimate (as a Python float) of how much of an upgrade has been completed, as a percentage.

Note – To determine when an upgrade is entirely complete see the `HOOK_OTA_UPGRADE_COMPLETE` section

10. *SNAP Connect* API Reference – Authentication Realms

The *realm* parameter is used by *SNAP Connect* when forming TCP connections. The value is passed to the authentication functions for accepting and connecting TCP. In versions of *SNAP Connect* prior to 3.2, this field would always take on the value "SNAPcom." With the addition of remote sniffers in *SNAP Connect* 3.2, the realm value can now be used to distinguish between regular TCP connections and remote sniffer TCP connections.

If desired, the *realm* parameter may be ignored in your client and server authentication functions and the login information for all connection types will be shared.

snap.REALM_SNAP – Standard SNAP over TCP Connections

This realm will be passed to authentication functions when `connect_tcp()` is used to create a connection between *SNAP Connect* instances. The parameter `sniffer_callback` should be set to None when creating a standard TCP connection.

See functions `accept_tcp()` and `connect_tcp()`.

snap.REALM_SNIFFER – Remote Sniffer connection over TCP

This realm will be passed to authentication functions when `connect_tcp()` is used to create a remote sniffer connection between *SNAP Connect* instances. The parameter `sniffer_callback` should be set to a callable function when creating a remote sniffer TCP connection.

This realm will only be used if `accept_sniffer()` has been called in addition to `accept_tcp()`.

See functions `accept_tcp()`, `accept_sniffer()`, and `connect_tcp()`.

11. *SNAP Connect* API Reference – Sniffer Descriptors

SNAP Descriptors are objects that are passed to local and remote sniffer callbacks. These descriptors represent packets that are received and transmitted by a *SNAP Connect* instance. This section lists the supported descriptors, their meanings, and their attributes.

The descriptors passed to sniffer functions are NOT over-the-air packets. These descriptors only represent packets that are received and transmitted over *SNAP Connect* serial and TCP connections.

Refer also to the description of the `accept sniffer()`, `connect tcp()`, and `start sniffer()` functions earlier in this manual.

NOTE – Because not every descriptor type supports every attribute, you will want to use the Python builtin function `isinstance(descriptor, descriptor_type)` to verify the descriptor type prior to accessing its attributes.

Common Attributes

All packet descriptors share several attributes in common.

- `raw` : A string of decrypted bytes representing the undecoded packet.
- `rx` : True if the packet was received by the *SNAP Connect* instance running the sniffer.
- `tx` : True if the packet was transmitted by the *SNAP Connect* instance running the sniffer.

snap.DataModeDescriptor – Data Mode Packets

Data mode descriptors represent received and transmitted (originated or forwarded) data mode packets.

See functions `data mode()` and `mcast data mode()`.

See HOOK_STDIN for how data mode packets are normally received.

- `data` : The data being carried by the packet.
- `dstAddr` : A three-byte string representing the immediate destination of the packet (subject to mesh routing) or a two-byte multicast group (for a multicast data mode packet).
- `final_dst_addr` : A three-byte string representing the final destination of the packet (unicast only).
- `isMulticast` : True if the packet is multicast or False if it is unicast.
- `seq` : The sequence number of the packet.
- `srcAddr` : A three-byte string representing the SNAP address of the original source of the packet.
- `Ttl` : The number of hops remaining for this packet (multicast only).

snap.MeshDescriptor – Mesh Routing Packets

Mesh descriptors represent received and transmitted mesh routing packets.

- `additionalAddresses` : An optional list of additional discovered/errored addresses.
- `hopLimit` : The number of hops remaining for this packet.
- `msgId` : A single character representing the type of mesh packet. ‘Q’ is used for route requests, ‘P’ is used for route replies, and ‘E’ is used for route errors.
- `source` : A three-byte string representing the immediate source of the packet.
- `originator` : A three-byte string representing the address of the node that created the packet.
- `originatorDistance` : The number of hops traveled from the originator.
- `target` : A three-byte string representing the target for the routing packet.

- `targetDistance` : A number representing the distance to the target.

snap.RawDescriptor – Undecoded Packets

Raw descriptors may show up in cases where a packet could not be decoded and broken into the correct components. A raw descriptor contains no additional information outside of the attributes common to all descriptors.

snap.RpcDescriptor – RPC Packets

RPC descriptors represent received and transmitted RPC packets.

See functions `rpc()`, `mcast rpc()`, and `dmcast rpc()`.

- `args` : A tuple of arguments to be passed to the RPC function.
- `dstAddr` : A three-byte string representing the immediate destination of the packet (subject to mesh routing) or a two-byte multicast group (for a multicast packet).
- `final_dst_addr` : A three-byte string representing the final destination of the packet (unicast only).
- `funcName` : Name of the RPC function to be called.
- `isMulticast` : True if the packet is multicast or False if it is unicast.
- `last_hop_addr` : A three-byte string representing the immediate source of the packet.
- `seq` : The sequence number of the packet.
- `srcAddr` : A three-byte string representing the original source of the packet.
- `ttl` : The number of hops remaining for this packet (multicast only).

snap.TraceRouteDescriptor – Trace Route Packets

Traceroute descriptors represent received and transmitted traceroute packets.

See function `traceroute()`. See HOOK_TRACEROUTE for information on how traceroute packets are received.

- `dstAddr` : A three-byte string representing the immediate destination of the packet.
- `elapsed` : The elapsed time in milliseconds to reach the target node. (This time does not include the first hop in the path, which will typically be trivial for serial connections but can be significant when *SNAP Connect* performs a traceroute over a TCP bridge.)
- `final_dst_addr` : A three-byte string representing the final destination of the packet.
- `isMulticast` : True if the packet is multicast or False if it is unicast.
- `msgId` : A single character representing the type of traceroute packet. ‘Q’ is used for traceroute queries and ‘P’ is used for traceroute replies.
- `orig_src_addr` : A three-byte string representing the original source of the packet.
- `records` : A list of traceroute records with ‘address’ and ‘linkQuality’ attributes.
- `srcAddr` : A three-byte string representing the immediate source of the packet.